TEKNILLINEN KORKEAKOULU
Tietotekniikan osasto

Tietojenkäsittelyopin laboratorio

Jari Kytöjoki

Anti-Virus-hakumoottorin suunnittelu ja toteutus kämmenmikrolaitteille

Diplomityö, joka on jätetty opinnäytteenä tarkastettavaksi diplomi-insinöörin tutkintoa varten Espoossa 10.6.2002.

Työn valvoja: Prof. Jorma Tarhio

TEKNILLINEN KORKEAKOULU **DIPLOMITYÖN TIIVISTELMÄ**

Tekijä: Jari Kytöjoki

Työn nimi: Anti-Virus-hakumoottorin suunnittelu ja toteutus kämmenmikrolaitteille

Päivämäärä: 3.6.2002 **Sivumäärä:** 60

Osasto: Tietotekniikan osasto

Professuuri: T-106, Ohjelmistojärjestelmät

Työn valvoja: Professori Jorma Tarhio

Työn ohjaaja: —

Tietoturva ja tietokonevirukset on otettu tässä diplomityössä tutkimuskohteeksi, kuinka voidaan suunnitella ja toteuttaa Anti-Virus-ohjelmiston hakumoottori jopa kämmenmikrolaitteille. Tällaiselle laitteistoalustalle toteutus ei ole helppoa rajoitettujen resurssien, prosessoritehon sekä käytettävyysvaatimusten takia, koska kyseessä on lähes tulkoon sulautettu järjestelmä.

Tässä diplomityössä esitellään useamman hahmon samanaikaiseen tarkkaan merkkijonotäsmäykseen tehokas menetelmä, joka on mahdollista tehdä pienellä muistimäärällä ja silti samalla etsiä tuhansia erilaisia, mutta samanmittaisia sormenjälkiä hyvinkin nopeasti. Lisäksi monia jo olemassa olevia täsmäysalgoritmeja arvioidaan ja tutkitaan, kuinka niiden parhaat puolet olisivat hyödynnettävissä ja miksi juuri niitä ei oteta sellaisenaan tämän diplomityön sovelluskohteen ongelman ratkaisemiseksi.

Teksti, josta eri hahmoja haetaan, tulee olemaan helposti useita satoja megatavuja, ja tämäkin asia huomioidaan jo aivan suunnitteluvaiheessa. Tälle esitetylle menetelmälle on löydettävissä useita niin uusia kuin vanhojakin sovelluskohteita.

Tässä diplomityössä ei kuvata, kuinka tietokonevirusten tartuntoja puhdistetaan yleisesti tai kuinka niille laaditaan tehokkaita heuristisia tunnistamismenetelmiä.

Hakusanat: Anti-Virus, hakumoottori, tarkka merkkijonotäsmäys, tarkka hahmonsovitus, algoritmit, monihahmo, etsintä, langaton tietoturva, kämmenmikrot, inkrementaaliset tietokannat

HELSINKI UNIVERSITY OF TECHNOLOGY **ABSTRACT OF THE MASTER'S THESIS**

Author:	Jari Kytöjoki
Title:	Design and Implementation of an Anti-Virus Scanning Engine for PDA devices
Date:	Jun 3, 2002 **Number of pages:** 60
Faculty:	Information Technology
Professorship:	T-106, Software Engineering (Information Processing Science)
Supervisor:	Professor Jorma Tarhio
Instructor:	—

Data security and computer viruses are taken as research subject in this Master's Thesis how we can design and implement a scanning engine for some Anti-Virus application even for PDA device. For this kind of hardware and platform it is not an easy task to do because of limited resources, processor performance and other usability requirements. These devices are mostly embedded systems.

In this thesis, it is presented an efficient multiple pattern search method for exact simultaneous matching. It is shown that this can be achieved with quite a little available free RAM while at the same time searching for a very large number, let's say over thousands of different patterns, that are equal in length i.e. fingerprints. In addition to that many already existing algorithms are evaluated and considered how the best sides of them are utilized and why we don't use them as is in this Master Thesis for a good solution base of the example application's problem field.

The text, which is used for matching the different patterns, can be very easily several megabytes and this fact will be noticed already during the design phase for the method. There are already several both new and old applications that could be using this presented method for solving the problems in a new way.

However, it is not described in this Master's Thesis how in general computer virus infections are cleaned and disinfected nor how heuristic rules are collected for detecting viruses in normal use cases.

Keywords: Anti-Virus, Scanning Engine, Exact String Matching, Algorithms, Multiple Pattern, Searching, Wireless Security, PDA, Incremental Databases

Alkulause

Tähän diplomityöhön liittyvä tutkimus- ja tuotekehitystyö on tehty vuosien 2001 ja 2002 aikana. Työ on ollut osa jatkuvaa tuotekehityshanketta, jossa parannetaan Anti-Virus-ohjelmistojen saatavuutta eri laitteistoille sekä uusille ympäristöille etenkin langattomille laitteille.

Haluan erityisesti kiittää työtä valvonutta prof. Jorma Tarhiota, joka kärsivällisesti jaksoi kannustaa ja ohjata minua lopullisen työn valmiiksi saamiseksi sekä samalla antoi ratkaisevia vihjeitä ja vinkkejä algoritmin edelleenkehittämisen suhteen tehokkaampaan suuntaan. Ilman näitä hedelmällisiä keskusteluja hänen kanssansa olisi algoritmi jäänyt huomattavasti huonommaksi nykyisestänsä. Suuri kiitos tästä kaikesta.

Lisäksi haluan vielä erikseen kiittää etenkin tietokantatoteutuksen testaamiseen osallistunutta Jarno Niemelää, joka sinnikkyydellänsä auttoi löytämään viimeisetkin pienet virheet. Pasi Lahti ja Pekka Usva ansaitsevat suuret kiitokseni liittyen työn lopulliseen kommentointiin työn saamiseksi julkaisukelpoiseksi. Tietysti haluan osoittaa kiitokseni myös koko mukana olleelle tiimille, jota ilman emme olisi saanet aikaiseksi loistavaa ja lajissansa ensimmäistä laitteessa sisäisesti suoritettavaa reaaliaikaista Anti-Virus–tuotetta kämmenmikroille.

Haluan kiittää ystäviäni ja työkavereitani heiltä saamastani tuesta. Erityisesti haluaisin osoittaa kiitokseni lähisukulaisilleni, jotka ovat antaneet minulle tukensa kaikkina vaikeinakin aikoina sekä jakaneet ilon hetket, te merkitsette minulle todella paljon, vaikka kiireiltäni en ole pystynyt Teitä tapaamaan niin usein kuin haluaisin.

Kirkkonummi, 3. kesäkuuta 2002

Jari Kytöjoki

Jari Kytöjoki

Puolukkakatu 5
FIN-44200 Suolahti
Suomi / Finland

+358-50-364 3770
Jari.Kytojoki@iki.fi
http://iki.fi/Jari.Kytojoki/

Sisällysluettelo

Lyhenteet ja käsitteet

AC	*Aho-Corasick*, Aho-Corasick-algoritmi
API	Application Programming Interface. Ohjelman kutsurajapinta.
ARM	Prosessori erityisesti kämmenmikrolaitteille. Kuuluisa virran käytön tehokkuudestansa (MIPS/W).
AV	Anti-Virus
big-endian	tavujärjestys muistissa prosessorin sisäisille tietotyypeille, jotka ovat tavua suurempia. Tässä tapauksessa tieto esitetään niin, että eniten merkitsevä tavu on ensin ja viimeisenä vähiten merkitsevä tavu. Usein puhutaan network-tavujärjestyksestä. Vastakohta little-endianille.
BIOS	Basic Input Output System. Laitteen ROM-muistiin tallennetut perustoimenpiteet ja rutiinit, joilla mahdollistetaan mm. käyttöjärjestelmän lataaminen kiintolevyltä tai levykkeeltä.
BM	*Boyer-Moore*, Boyer-Moore-algoritmi
BMH	*Boyer-Moore-Horspool*, Boyer-Moore-Horspool-algoritmi
BMHHS	*Boyer-Moore-Horspool-Hume-Sunday*, Boyer-Moore-Horspool-Hume-Sunday-algoritmi
BS	Binary Search, binäärihaku.
CISC	Complex Instruction Set Computer, laajan käskykannan tietokone, vastakohta RISC:lle
CPU	Central Processing Unit. Keskusyksikkö, prosessori.
CRC	Cyclic Redundancy Check. Eräs tavallisin tarkistussumman laskentamenetelmä; virheentunnistusmenetelmä.

EICAR	European Institute of Computer Anti-virus Research. Eurooppalainen tietokonevirusten tutkimiseen keskittynyt tutkimuslaitos.
EEPROM	Electrical Erasable Programmable Read Only Memory. Sähköisesti tyhjennettävä lukumuisti.
EPOC	Symbian-käyttöjärjestelmän (*Symbian OS*) aikaisempi nimitys ennen sen uutta 6-sarjaa.
ER5	EPOC Release 5 mainitaan kun puhutaan Symbian-käyttö-järjestelmän (ER6) edeltävästä ajasta.
GSM	Global System for Mobile communications
hahmo	Merkkijono (Pattern), jota halutaan etsiä osajonona toisesta merkkijonosta eli tekstistä ts. sille halutaan tehdä täsmäys ko. tekstissä. Ks. myös teksti.
I/O	Input/Output. Syöttö/Tulostus.
kB	Kilotavua, kilo bytes; 1024 tavua.
KMP	*Knuth-Morris-Pratt*, Knuth-Morris-Pratt-algoritmi
little-endian	tavujärjestys muistissa prosessorin sisäisille tietotyypeille, jotka ovat tavua suurempia. Tässä tapauksessa tieto esitetään niin, että vähiten merkitsevä tavu on ensin ja viimeisenä eniten merkitsevä tavu. Vastakohta big-endianille.
MIPS	Million Instructions Per Second, Miljoonia käskyjä sekunnissa.
PDA	Personal Digital Assistant eli henkilökohtainen muistikirja.
PROM	Programmable Read Only Memory. Kertaohjelmoitava lukumuisti.
RAM	Random Access Memory. Hajasaantimuisti.

RISC	Reduced Instruction Set Computer, vähennetyn käskykannan tietokone. Vastakohta CISC:lle.
ROM	Read Only Memory. Lukumuisti.
RK	*Rabin-Karp*, Rabin-Karp-algoritmi
SDK	Software Development Kit eli ohjelmistonkehityspaketti
SMS	Short Message System, 160 merkin viestinvälitysjärjestelmä GSM-järjestelmässä.
Snort	Open Source Network Intrusion Detection System. http://www.snort.org/
teksti	Merkkijono (String), mistä hahmoa yritetään tehdä täsmäys. Olkoon $S = s_1s_2...s_n \in \Sigma^+$ merkkijono (teksti) ja $P = p_1p_2...p_m \in \Sigma^+$ toinen merkkijono (hahmo) aakkostossa Σ, $\lvert \Sigma \rvert = c$. Sisältääkö S osajononaan P:n ts. onko S muotoa S = S'PS" joillakin jonoilla S', S"? Huom. Yleensä tavallisissa sovelluksissa m << n ja n voi olla hyvinkin suuri.

1. Johdanto

Monet Anti-Virus-ohjelmistojen tuotekehitysalueella kilpailevat yritykset ovat vielä tällä hetkellä arvioineet, että täysin toimivan virushakumoottorin toteuttaminen kämmenmikroille on käytännössä katsoen melko mahdotonta tämänhetkisillä laitteistoilla. Ainakin suuria ongelmia tulisi olemaan sekä ohjelmiston suorituskyvyn että nopeuden kuin myös muistin käytön suhteen.

Toinen suuri syy, miksi useat Anti-Virus-tuotteiden valmistajat eivät ole olleet kiinnostuneita suojaamaan kämmenmikrolaitteita uusilla Anti-Virus-ohjelmistoilla, on luonnollisesti ollut markkinoiden pienuus sekä vielä tällä hetkellä puuttuvien, mutta todellisten, uhkakuvien toteutuminen käytännössä. Tilanne on kuitenkin se, että se kuka saa ensimmäisenä tuotteen valmiiksi, hallitsee ainakin aluksi markkinoita todellisten ongelmien alkaessa saavuttaa vakavampia mittasuhteita. Mutta asiaan on suhtauduttava hyvinkin vakavasti jo nyt ja lienee lähinnä vain ajan kysymys, milloin nykyiset ja tulevat virusten kirjoittajat alkavat toden teolla kiinnostua näistä uusista laitteista havaitessansa virusten kirjoittamisen olevan niille mahdollista aivan samalla lailla kuin muillekin tietokoneille hyödyntäen havaittuja tietoturva-aukkoja.

Tämän projektin pohjalta oli pyrkimys myös hakea arvokasta kokemusta AV-ohjelmistojen toteuttamisesta myös muille kämmenmikrolaitteille, mitkä olisivat yleiset ongelmakohdat ja kuinka ne voitaisiin ratkaista mahdollisimman hyvin.

2. Taustaa

2.1 Projektin lähtökohdista

Tämä Anti-Virus-hakumoottorin suunnittelu- ja toteutusprojekti tuli heti alusta alkaen olemaan lähes yhden miehen projekti työssä vaadittavan suuren tietotaidon takia. Projektin aikana tulisi saada valmiiksi ensimmäinen laatuansa oleva hakumoottori, joka olisi suunniteltu erityisesti kämmenmikrolaitteita silmällä pitäen. Työ tulisi sisältämään ainakin seuraavat osakokonaisuuksien suunnittelun ja toteutukset.

- haku- eli täsmäysalgoritmi
- rajapinnan suunnittelu hakumoottorin käytölle ylemmältä tasolta
- tietokantatiedostoformaatit sekä päätietokanta että inkrementaaliset päivitykset
- tietokantojen virustunnisteiden skriptikuvauskielen
- työkalut tietokantojen tekemiseen, käsittelyyn, analysointiin jne.
- tarvittava dokumentaatio

Projektissa tulisi olemaan siis haastetta kerraksensa heti alusta alkaen. Aivan alkuvaiheessa olikin suurena apuna tämän diplomityön tekijän vankka kokemus ohjelmoinnista rajoitetulla muisti- ja prosessoritehomäärällä ja onneksi aikoinaan jo 80-luvulla oli harrastuksena ja mielenkiinnon kohteena ohjelmointi Commodore 64 -tietokoneella konekielitasolla, jolloin oppi todella käyttämään vähäisen muistimäärän viimeistä bittiä myöten sekä optimoimaan käskyt niiden viemien kellojaksojen tarkkuudella. Myös kokemus 90-luvun alkupuolella toiminnasta tietokonedemojen ja 4 Kb:n introjen parissa Intelin prosessoriympäristössä auttoi työn toteuttamisessa etenkin juuri sen alkuvaiheessa. Sulautetut järjestelmät ovat lisäksi aina olleet yksi mielenkiinnon kohde siitä lähtien, kun asiaan pääsi tutustumaan lähimmin opiskelujen kautta, mutta mitään käytännön kokemusta suuremmista ja todellisista sulautetuista järjestelmistä tämän diplomityöntekijällä ei ole ollut.

Viime aikoina monet Anti-Virus-ohjelmistojen valmistajat ovat jo julkistaneet omia ohjelmistojansa, joita he ehkä hieman erheellisestikin mainostavat, pystyvän tekemään tutkimisen myös kämmenmikrolaitteiden osalta varsinaisen laitteen sisällä tai ainakin tällaisen kuvan tavallinen kuluttaja saa hyvin nopeasti tutkittuaan ko. ohjelmistojen esitteitä. Näiden ohjelmistojen toimintaperiaate ei kuitenkaan ole ollut itse laitteessa tapahtuva reaaliaikainen tutkiminen (*real-time on-device-scanning*), vaan tutkiminen on tapahtunut laitteen synkronoinnin yhteydessä työasemaan, jolloin tutkimistyö on tapahtunut itse asiassa työaseman ohjelmistolla (*"off-device-scanning"*) eikä suoraan

PDA-laitteen sisällä sen omalla prosessorilla. Muut AV-ohjelmistojen kehittäjät ovat kuitenkin maininneet julkaisevansa mahdollisimman pian myös tuotteita, jotka tekevät todellisen tutkimisen laitteessa jopa reaaliaikaisesti sekä tukevat tietokantapohjaista tunnistetietojen päivittämistä.

2.2 Troijan hevoset ja tietokonevirukset

Tietoturva-alalla termi tietokonevirus ei ole kovin yksiselitteinen. Virusohjelma voi olla toiminnaltaan tuhoisa, vitsikäs tai jopa hyödyllinenkin, mutta oleellista on sen leviäminen käyttäjältä lähes salassa. Tavallinen käyttäjä pitää usein tietokonevirusta haitallisena ohjelmana tai sen osana, joka pyrkii tekemään pahoja tekoja käyttäjän tietokoneessa poistamalla tiedostoja ja onpa olemassa jopa uskomuksia, että eräät virukset saattaisivat aiheuttaa laitteistoille pysyviä fyysisiä vaurioita, esim. muuttamalla näytönohjaimen virkistystaajuuksia monitorille sopimattomaksi tai ohjaamalla kiintolevyn lukupäitä tekemään hurjia suorituksia. Tosiasia on kuitenkin, ettei tällaisia pysyviin fyysisiin vaurioihin pystyviä viruksia ole kuitenkaan juuri ilmaantunut ja pahimmat tuhot ovat lähinnä sellaisia kuin koneen emolevyn BIOS:in EEPROM:in tyhjentäminen, jolloin tällainen emolevy yleensä muuttuu käyttökelvottomaksi. Valittavasti eräät emolevyjen valmistajat eivät aikoinaan viitsineet kiinnittää tarpeeksi huomiota BIOS:in päivitystekniikoihinsa, jolloin viruksille tuli mahdolliseksi tämän tapaisten ongelmien aiheuttaminen. Tunnetuin BIOS:in tyhjentävä virus on nimeltänsä CIH alias Tšernobyl, joka aktivoituu vuosittain huhtikuun 26. päivänä.

Yleisesti ottaen aivan tavallisen käyttäjän määritelmä on varsin epätarkka ollakseen hyvä määritelmä tietokonevirukselle. Eräs usein käytetty määritelmä perustuu Edward Amoroson esittämään jaotteluun, jossa hän jakaa haitalliset ohjelmat kahteen pääluokkaan *Troijan hevosiin* ja varsinaisiin tietokoneviruksiin [3]. Yleisesti ottaen hänen mielestänsä troijalaiset ovat aivan kuin mikä tahansa muukin hyödyllinen tavallinen tietokoneohjelma, joka kuitenkin tekee jotain aivan odottamatonta ja ei-toivottuja toimenpiteitä. Tietokoneviruksia hän pitää troijalaisten alalajina, joka pystyy levittäytymään omaa koodiansa kopioimalla yhä uudelleen aivan samoin kuin oikeat biologisetkin virukset toimivat. Laajimmillaan Amoroso käsittelee jopa ohjelmointivirheiden tahallisuutta ohjelmistoissa kuuluvan eräiltä osin troijalaisten määritelmän joukkoon.

Kuitenkin ehkäpä paras määritelmä tietokonevirusten luokittelussa virusten ja troijalaisten välillä on islantilaisen tietokonevirustutkijan ja ohjelmoijan Friðrik

Skúlasonin jaottelu noin 90-luvun alkupuolelta [8]. Hän on tunnetun ilmaisen F-Prot Anti-Virus-ohjelman alkuunpanija ja on toiminut alalla jo lähes 15 vuotta ja hänen ohjelmansa on edelleen aktiivisen kehitystyön alla ja sitä pidetään yleisesti maailmalla yhtenä parhaimmista AV-ohjelmista niin tunnistamisen kuin puhdistamisen suhteen [12]. Alunperin ohjelma oli saatavana vain MS-DOS-ympäristöön, mutta nykyisin ohjelma on saatavilla useammille eri käyttöjärjestelmille. Tosin tunnistaminen virusten suhteen on suurimmalta osin rajattu lähinnä Intelin prosessoriarkkitehtuurin mukaisiin ohjelmiin.

Skúlasonin määritelmä [8] on seuraava, joka on tämän diplomityön tekijänkin mielestä yksi parhaimmista, koska se on tarkasti rajattu.

1. Virus on ohjelma, joka kykenee kopiomaan itseänsä eli replikoitumaan niin että syntyneet uudet ohjelmat saattavat olla hieman muuttuneita, jolloin puhutaan mutaatioista. Tämä erottaa hänen mukaansa viruksen troijalaisista ja muista ei-replikoituvista haitallista ohjelmista (malware).
2. Viruksen lisääntyminen kopioitumalla on tarkoitushakuista eikä vain jonkin tapahtuman sivutulos. On tärkeää esim. sulkea pois sellaiset levykkeenkopiointi-ohjelmat, jotka vain kopioivat saman levykkeen sisällön toiselle, jolla ne itsekin alunperin olivat.
3. Ainakin jotkut viruksen jälkeläisistä ovat myös toimivia viruksia. On mahdollista, että osa uusista jälkeläisistä ei pysty lisääntymään tai ei toimi ollenkaan. On tärkeää erottaa toimivat ja aiotut virukset sellaisista, jotka eivät kuitenkaan toimi esim. pienen yksinkertaisen ohjelmointivirheen seurauksena.
4. Viruksen tulee liittyä osaksi isäntäohjelmaa siten, että ohjelman suoritus saa aikaan sekä isäntä- että viruskoodin suorituksen samaan aikaan. Tällä erotetaan tietokonevirukset tietokonemadoista, mutta samalla määritelmä on oltava niin laaja, että se kattaa vertaisvirukset (companion viruses) että dokumenttien makrovirukset.

2.2.1 Yleistä

Tässä diplomityössä kerrotaan vain hyvin lyhyesti ja yleisellä tasolla tietokone-viruksista. Mikäli kohdelaitteelle ja sen käyttöjärjestelmälle on olemassa todistetusti tietokoneviruksia on myös aina olemassa ikävä mahdollisuus joutua sellaisen viruksen kohteeksi. Tietokonevirukset ovat todella vaarallisia verrattuna muihin useimpiin tietoturvariskeihin, sillä vaikka kaikki tiedot muistettaisiinkin varmuuskopioida säännöllisesti on silti vaarana, että virus pääsee saastuttamaan myös vähitellen varmuuskopiot tehden ne usein täysin käyttökelvottomiksi. Toki säännöllisesti

suoritettava varmuuskopiointi pienentää täystuhon mahdollisuutta ja useimmissa tapauksissa tietoa pystytään näin pelastamaan, mutta silloin tällöin tulee viruksen iskiessä tilanne, että myös varmuuskopiot saattavat olla käyttökelvottomia viimeisten viikkojen ajalta. Ja kukapa sitä usein pitääkään kuukausien takaisia varmuuskopioita tallessa?

Puutteellisesti toteutetussa virussuojauksessa on se ikävä puoli, että hyvin ja tehokkaasti suunniteltu virus voi huomaamattomasti ja erittäin äkkiä levitä melko laajalle alueelle samalla mahdollisesti pilaten sekä maineen että luottamuksen sen henkilön tai organisaation osalta, jonka järjestelmiin tällainen virus pääsee hyökkäämään sisään. Virus voi toimia salakavalan hitaasti taustalla tai aktivoitua hieman myöhemmin vakavammin seurauksin, jolloin saattaa olla hyvinkin vähän tehtävissä tärkeiden tietojen pelastamiseksi.

Vaikka todellisen vakavan virustartunnan vaara onkin melko pieni olisi silti hyvä varautua uhkakuviin niin hyvin kuin se vain on mahdollista.

2.2.2 Yleisimmät virustyypit

Tällä hetkellä tietokonevirukset ovat lähinnä olleet vain kaikkien suosituimpien ja yleisimpien käyttöjärjestelmien ongelmana ja erityisesti ongelma koskee Microsoftin kehittämiä käyttöjärjestelmiä, jotka ovat olleet MS-DOSiin pohjautuvia. Mutta myös muille käyttöjärjestelmille on kehitelty viruksia eikä ollenkaan sovi unohtaa mm. Linuxille kehitettyjä, sillä Linuxia kuten myös muita Unix-järjestelmiä pidetään paremmin toteutettuina tietoturvan näkökulmasta. On olemassa jopa muutama sellainen virus joka toimii Intelin x86-prosessoreilla sekä Windows- että Linux-ympäristöissä. Myös aikoinaan 1980-luvun lopulla ja 1990-luvun alussa Commodoren Amiga–tietokoneperhe oli kuuluisa sitä järjestelmää koskevista omista tietokoneviruksistansa ja niiden aiheuttamista epidemioista.

Tavallisesti tietokonevirukset on jaettu karkeasti seuraavaan viiteen tyyppiin eli

- makrovirukset
- komentosarjavirukset
- tiedostovirukset
- käynnistyssektorivirukset
- vertaisvirukset

Makrovirukset ovat tavallisesti kirjoitettu jonkin sovelluksen makro- tai muulla vastaavalla skriptikielellä ja tavallisimmin tämäntyyppiset virukset hyödyntävät kyseisen kielen tietoturva-aukkoja tai suunnitteluvirheitä. Makrovirukset leviävät vaarallisen tehokkaasti, koska ihmisillä on usein tapana vaihdella keskenänsä suosituimmilla ohjelmilla tuotettuja dokumentteja ja valitettavasti mm. Microsoftin Word-tekstinkäsittelyohjelmisto kuin myös koko muu Office-tuoteperhe on tästä ikävähkönä esimerkkinä.

Komentosarja- eli skriptivirukset muistuttavat hyvin paljon makroviruksia, mutta niiden viimeaikainen runsas lisääntyminen on aiheuttanut kyseisen virustyypin eriyttämisen omaksensa. Komentosarjavirukset eivät kuitenkaan tarvitse käyttöjärjestelmän lisäksi muuta sovellusta levitäksensä, sillä ne käyttävät leviämiseen käyttöjärjestelmän tarjoamia mahdollisuuksia kuten esim. Windows 98, ME ja 2000 järjestelmiin usein niin automaattisesti asennettua VBS-kieltä (Visual Basic Script), joka muistuttaa hyvin paljon Microsoft Office -tuotteiden käyttämää VBA-kieltä (Visual Basic for Applications).

Tiedostovirukset ovat lähes se klassisin virustyyppi. Nimittäin kuten nimi kertoo nämä virukset tavallisimmin tarttuvat vain ohjelmatiedostoihin, mutta toisinaan ehkäpä yksinkertaisten ohjelmointivirheiden vuoksi ne saattavat saastuttaa ja korruptoida myös muuntyyppisiä tiedostoja vahingossa. Tiedostovirus etsii usein hyvin erilaisin kriteerein kohteita viruksen lisäämiseksi uuteen isäntäohjelmaan ja tavallisimmin sen loppuun tai muuten vain sen lähes käyttämättömiin osiin tiedostossa. Virus muuttaa isäntäohjelman suorituksen alkukohtaa siten, että se varmistaa saavansa ohjelman suorituksen alun itsellensä ennen varsinaisen isäntäohjelman suorituksen aloittamista. Tällä aktivointikoodilla virus sitten hallitsee itsensä levittämistä uusiin tiedostoihin aikaisemmin puhtaina oleviin järjestelmiin. Tiedostovirukset hyödyntävät usein hyvin tehokkaasti käyttöjärjestelmän toteutuksessa olevia vakavia tietoturva-aukkoja ja -ongelmia.

Käynnistyssektorivirukset ovat aikojen kuluessa olleet tiedostovirusten ohella eräs ensimmäisistä tietokonevirustyypeistä. Nämä virukset saastuttavat kiintolevyn tai levykkeen käynnistyslohkosektoreita mahdollistaen itsellensä suorituksen ennen varsinaisen käyttöjärjestelmän tietojen lataamista. Näin toimien virus saa haltuunsa järjestelmän ainakin tiettyyn vaiheeseen asti ja pyrkii piilottautumaan järjestelmään erilaisin naamioitumis- eli *stealth*-tekniikoin. Usein tällaiset virukset järjestelevät

laitteen BIOS-kutsut uudelleen itsensä kautta, jolloin ne pystyvät tarkkailemaan järjestelmän toimintoja ja tarvittaessa leviämään eteenpäin samanaikaisesti. Käynnistyssektorivirukset voivat periaatteessa tartuttaa mitä tahansa käyttöjärjestelmää (DOS, Windows, Linux, NetWare jne.) käyttävän PC:n. Mikäli tällaisessa tilanteessa tartunnan saaneen PC:n käyttöjärjestelmä ei olekaan DOS-pohjainen, ei tietokone tavallisesti käynnisty lainkaan ja järjestelmän palauttaminen toimintakelpoiseksi saattaa olla hyvinkin vaikeaa.

Vertaisvirukset eli rinnakkaisen tiedoston luovat virukset käyttävät eräitä käyttöjärjestelmän erikoisuuksia hyväksensä, kuten esim. MS-DOSissa .com-päätteiset tiedostot ajetaan aina ennen .exe-päätteisiä, mikäli ohjelma käynnistetään sen lyhyemmällä nimellä ilman tiedostopäätettä. On myös mahdollista, että virus luo vastaavasta .exe-tiedostosta samannimisen tiedoston .com-päätteellä ja samalla tiedosto merkitään piilotiedostoksi, jolloin se ei näy suoraan tavallisissa hakemistotiedostoluetteloissa suoraan. Toinen tapa on muuttaa käyttöjärjestelmän PATH-ympäristömuuttujaa niin, että tiedostoja ajetaankin ensin aivan toisesta hakemistosta kuin ehkä alunperin onkaan ollut tarkoitus. Aktivoitunut vertaisvirus ajaa aluksi varsinaisen isäntäohjelman, mutta kun sen suoritus lopetetaan siirtyy suoritus varsinaiselle virusohjelmalle.

Useimmiten virukset ovat rajoittuneita vain kohdekäyttöjärjestelmänsä saastuttamiseen, mutta makro- ja komentosarjavirukset pystyvät periaatteessa toimimaan muillakin käyttöjärjestelmillä, jos vain niihin on toteutettu vastaava tuki, esim. Apple Macintoshin Office-tuotteet.

2.2.3 Kehittyneemmät virustyypit

Muutamia vuosia sitten näytti siltä kuin Anti-Virus-ohjelmistot olisivat saamassa voiton tavallisista viruksista kattavan tarkistussummalaskennan esim. CRC- ja sormenjälkietsinnän kautta. Mutta myös virusten kirjoittajat etenivät tekniikoissansa eteenpäin ja vastasivat haasteisiin uusilla luomuksillansa. Tällaisia uusia virustyyppejä on nk. piilovirukset, jotka muuttavat käyttöjärjestelmän kutsuja niin, ettei niitä ole mahdollista havaita suoraan sitä kautta, sillä ne valehtelevat muille ohjelmille tietoja niiden tiedostojen kohdalla, joihin kyseinen virus on tarttunut. Tämän teeskentelyn kautta tiedostot näyttävät edelleen olevan ikään kuin aivan puhtaita.

Toinen päätyyppi on itsensä salaavat ja muuntuvat eli polymorfiset virukset, joiden tunnistaminen on erittäin hankalaa. Itse asiassa siihen on olemassa vain yksi varma ratkaisu, mikä tarkoittaa ohjelmatiedoston suorituksen emuloimista konekielitasolla saakka. Tällöin vaatimuksena on täysiverisen nopean emulaattoriohjelmiston luominen ja tätä emulaattoria on voitava hyödyntää ongelmallisten tiedostojen aukipurkamiseen. Polymorfiset virukset sisältävät useita tasoja usein käsittämättömän näköisiä käskyjä, jotka ovat järjestettyjä niin, ettei ne joko tee mitään järkevää ts. ovat käskykannan hyödyttömiä käskyjä kuten nollan lisääminen yms. tai sitten osallistuvat varsinaisen viruskoodin purkamiseen vähitellen useamman iterointitason kautta. Hankalimpien useampitasoisten polymorfisten virusten avaaminen saattaa vaatia miljoonien käskyjen suorituksen emulointia. Polymorfisten virusten tartunnat uusiin tiedostoihin eivät juuri koskaan ole samanlaisia ja se, jos mikä, tekee niiden tunnistamisesta erityisen ongelmallista.

Polymorfisista viruksista on olemassa myös alaluokka metamorfiset virukset, jotka osaavat korvata ohjelman konekielikoodia toisella vastaavanlaisella samanmittaisella käskysarjalla aina uuden sukupolven välillä saman lopputuloksen saavuttamiseksi. Esim. rekisteristä toiseen kopiointia ei tehdäkään suoraan vaan se tehdään pinon kautta vaikkakin hieman hitaammalla tavalla.

Virusten kirjoittajat ovat luoneet myös vastaviruksia, joiden ensisijaisena kohteena on ollut hyökätä tiettyjä Anti-Virus-ohjelmistoja vastaan esim. tuhoamalla ko. ohjelmiston virustunnistetietokantoja.

Monimuotoiset virukset osaavat useimpia tavallisimpia virusten leviämistekniikoita ja ne hyödyntävätkin niitä tehokkaasti tartuttaen niin käynnistyssektorit, ohjelmatiedostot kuin myös dokumenttien makrot. Monimuotoisilla viruksilla saattaa periaatteessa olla paremmat mahdollisuudet selvitä kattavasta virusten puhdistusoperaatioista huolimatta.

Näiden kehittyneempien virusten löytäminen on vaikeampaa ja usein ongelmana onkin ollut järjestelmän käynnistäminen uudelleen niin, ettei yksikään tällainen virus ole vielä ehtinyt kaapata itsellensä kaikkia käyttöjärjestelmän toimintoja haltuunsa, jotta kyseisten virusten tunnistaminen ja poistaminen voitaisiin edes toteuttaa.

2.2.4 Virustentorjuntaohjelman toiminnan nopea testaaminen

Kaikkien nykyaikaisten virustentorjuntaohjelmistojen toiminnan testaaminen on hyvin nopeaa ja yksinkertaista sen suhteen, onko tunnistaminen päällä, ettei se vahingossa ole kytkettynä pois käytöstä esim. hitauden vuoksi. Yhteistyössä on luotu eräänlainen testitiedosto, joka ei ole varsinaisesti oikea virus, mutta sen tunnistamisessa toimitaan kuin se olisi oikea. Tämän testitiedoston on luonnut EICAR (European Institute of Computer Anti-virus Research), joka on eurooppalainen tietokonevirusten tutkimiseen keskittynyt tutkimuslaitos. Nimeltänsä tiedosto on yksinkertaisesti EICAR-testitiedosto ja itse asiassa tällaisen testitiedoston luominen on hyvin helppoa. Riittää kun leikkaa ja liimaa seuraavan rivin omaksi tiedostoksensa ja kokeilee sen tunnistamista omalla ohjelmistollansa.

```
X5O!P%@AP[4\PZX54(P^)7CC)7}$EICAR-STANDARD-ANTIVIRUS-TEST-FILE!$H+H*
```

Lisätietoa aiheesta löytyy EICARin kotisivuilta http://www.eicar.org/.

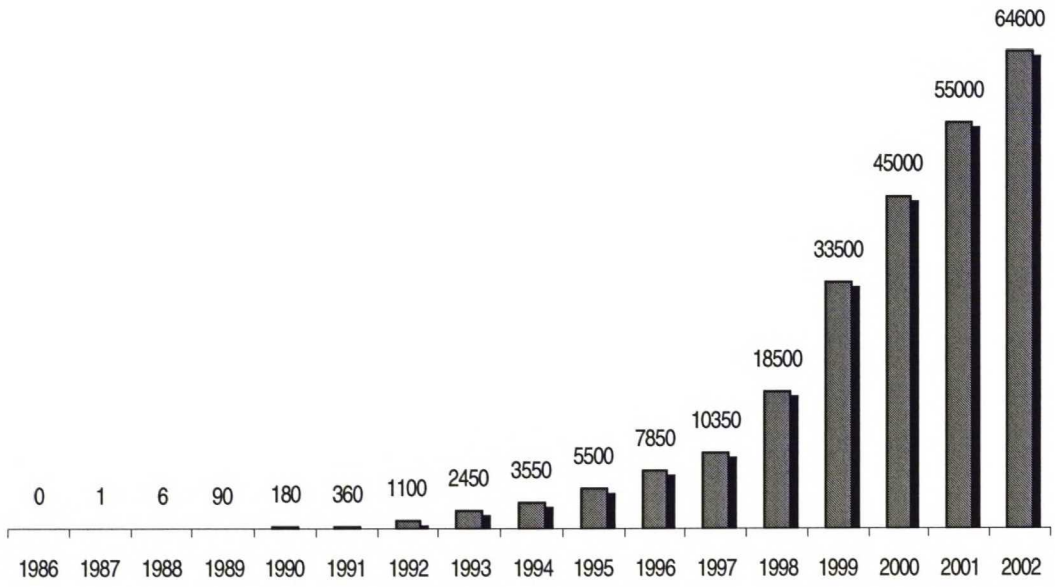
2.2.5 Tietokonevirusten lukumäärän kehitys 1986-2002

Nykyisissä yleisimmissä käyttöjärjestelmäympäristöissä erilaisten tietokonevirusten lukumäärän kehitys on ollut eksponentiaalista aivan viime vuosiin asti. Kehittyneempien torjuntatekniikoiden ansiosta virusten lukumäärän kasvu ollaan vähitellen saamassa hyvään hallintaan, tosin samalla uudet virukset ovat lähes aina hankalampia tunnistaa kuin edeltäjänsä. Alkuvuonna 2002 tilanne on ollut seuraavanlainen.

Ohjelmatiedostoviruksia eli PC-binäärivirukset		yli 54000
• DOS	53500	
• Windows 3.x	15	
• Windows 95	500	
• Windows NT	70	
Makro- ja komentosarjavirukset		yli 10500
• Word	6000	
• Excel	1500	
• PowerPoint	5	
• Muut (Java, BAT, Script etc)	3000	

Muut	alle 200
• Macintosh	50
• Linux	127
• Symbian OS / EPOC	6 troijalaista
• Palm OS	3 troijalaista, 1 virus
Yhteensä	yli 64600

Yleisesti ottaen tällä hetkellä maailmalla tavataan yleisimmin vain alle 1000 erityyppistä tietokonevirusta vähemmän tai enemmän aktiivisessa liikkeessä. Suurin osa vanhemmista viruksista on saatu siivottua pois ja lisättyä Anti-Virus-ohjelmien tunnistetietoihin, joten niiden aktiivista leviämistä on pystytty tehokkaasti estämään.



Kuva 1. Tietokonevirusten lukumäärän kehitys 1986-2002

Suuri hyppy virusten lukumäärässä erityisesti vuosien 1998-1999 välillä johtuu lähinnä uusien vaikeiden polymorfisten virusten ilmaantumisesta Windowsin 32-bittisiin ympäristöihin ja kesti jonkin aikaa, ennen kuin AV-ohjelmistojen hakumoottoreiden emulaattorikomponentit oli saatu päivitettyä niin, että ne pystyivät emuloimaan myös tällaisia hankalia viruksia. Samanaikaisesti sattui lisäksi useita pienempiä makrovirusten epidemioita.

2.2.6 Tulevaisuuden uhkakuvat

Tämän päivän tehokkaimmat kämmenmikrot alkavat olla jo suorituskyvyltänsäkin sellaisia, kuin mitä tavallisesti olivat kotitietokoneet vain noin viisi vuotta sitten. On oletettavaa, että kunhan kämmenmikrolaitteistot yleistyvät myös virusten kirjoittajat kokevat haasteena kirjoittaa uusia näille laitteille tarkoitettuja viruksia ja samalla he todennäköisesti haluavat kokeilla langattomuuden tuomia etuja uusien virusten levittämistapojen löytämiseksi.

Onkin vain ajan kysymys, kun ensimmäinen virusepidemia on ikävästi totta jollain nykyisistä kämmenmikrolaitteista. Itse asiassa seuraavanlaisia tiedotuksia on jo alkanut ilmaantua virusten kirjoittajien suosimille WWW-sivuille. *"Let's go to work. We are starting Cell Phone Virus Challenge. Any contribution welcomed (the more funny solution, the better). Deadline has not been set"*. Suora lainaus eräältä virusten kirjoittajien suosimalta WWW-vastarintauutispalvelimelta, *http://virus.cyberspace.sk* kesäkuussa 2000.

2.3 Symbian-käyttöjärjestelmä

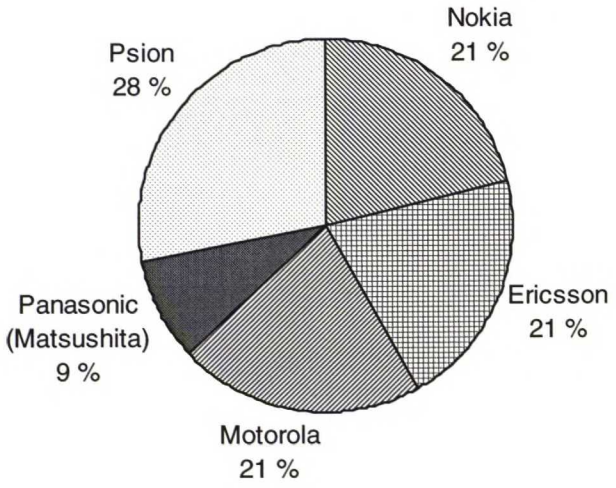
2.3.1 Yleistä

Symbian-käyttöjärjestelmä (Symbian OS) tunnettiin vielä kevään 2001 loppuun saakka yleisesti EPOC-alustana (EPOC Platform). Lopulta nimenmuutos käyttöjärjestelmälle tuli kuitenkin ajankohtaiseksi versionumeron muuttuessa 5-sarjasta 6-sarjaan ja yleensä tuolloin puhuttiinkin ER5:stä eli EPOC Release 5:stä. Muutoksella haluttiin mitä ilmeisemmin viestittää uuden käyttöjärjestelmän todellisesta ominaisuuksista täysivertaisena kilpailukykyisenä käyttöjärjestelmänä. Symbian ja sen edeltäjä EPOC ovat Symbianin kehittämiä kevyille kannettaville PDA-laitteille suunnattu käyttöjärjestelmä.

Symbian on avoin käyttöjärjestelmä kuten esim. myös Linux on. Näin ollen sille on helpompi kehittää kaikenlaista uutta ohjelmistoa kuin monille suljetummille käyttöjärjestelmille. Esimerkiksi Microsoftin käyttöjärjestelmien eräät osat ovat hyvin huonosti dokumentoituja ja jos on tarve tehdä jokin sovellusohjelma, jonka toteuttamiseksi vaaditaan lisätietoa näistä dokumentoimattomista käyttöjärjestelmän osista, joudutaan hyvin usein turvautumaan nk. reverse-engineering-tekniikoihin eli tutkitaan jo olemassa olevia Microsoftin omia sovelluksia ja ajureita, vaikka käytännössä ohjelmistojen lisenssiehdot usein kieltävätkin tällaiset toimenpiteet.

Symbian-käyttöjärjestelmälle on tarjolla ilmaisia SDK:ita, joilla on helppo päästä alkuun ohjelmistonkehityskokeiluissa.

Itse Symbian-yritys muodostui heinäkuussa 1998 ja sen omistavat maailman johtavat matkapuhelinvalmistajat seuraavin prosenttiosuuksin [16].



Toimistoja Symbianilla on Isossa-Britanniassa, Ruotsissa, Japanissa ja Yhdysvalloissa. Työntekijöitä on yhteensä noin 700. Historiallisista syistä Symbianilla on vahvat juuret ja yhteydet Psion Ltd. -yhtiöön.

Symbian on erityisesti suunnattu mobiililaitteille ja on siten erinomainen valinta kämmenmikrolaitteen käyttöjärjestelmäksi.

Symbian-käyttöjärjestelmän pääominaisuudet [16]

- Kirjoitettu suurimmaksi osaksi C++:lla, toteutukseltansa hyvin oliopohjainen ja tapahtumaohjattu eli eventtilähtöinen (event driven).
- Robusti mikro-Kernel suunnittelun pohjana
- Todellinen, oikea keskeyttävä moniajokäyttöjärjestelmä (pre-emptive multi-tasking system)
- Sisäänrakennettu hyvä virrankulutuksen seuranta ja hallinta
- Ajettavissa suoraan ROM-muistista
- Valmis tuki mm. Javalle, infrapunatiedonsiirrolle (IrDA) jne.
- Joustava käyttöliittymäkehityspohja

2.3.2 Muistinhallinnasta lyhyesti

Kämmenmikrolaitteet, jotka käyttävät mm. Symbiania, ovat yleisesti sulautettuja järjestelmiä, joissa muistinhallintaan tulee kiinnittää suurta huomiota. Laitteen käyttöjärjestelmä ja sovellukset voivat olla yhtäjaksoisesti käytössä kuukausia ilman katkoksia tai uudelleenkäynnistämisiä.

Lisäksi muistin loppumisesta aiheutuvat virhetilanteet ovat melko yleisiä ja hyvinkin todennäköisiä verrattaessa esim. valtavan virtuaalimuistin varassa toimivaan normaaliin työasemaan [2]. Tällaisissa kämmenlaitteissa on usein muistia vain muutamia kymmeniä megatavuja ja mitään kovalevyjä tai muita massamuistilaitteita ei tunneta eikä niitä ole myöskään käytettävissä muuten kuin varmuuskopioimalla tietoja jollekin työasemakoneelle. Sama muistiavaruus jaetaan tavallisesti sekä sovellusten käyttömuistina että myös tiedostojen tallennustilana ikään kuin emuloiden virtuaalista levyjärjestelmää.

Muistinhallinnalle teettää suurta työtä myös se, että kuinka pitää muistin roskaantuminen pienenä ja kuinka hallita muistin fragmentoituminen, mikäli sovellus ei jostain syystä vapauta ajallansa varattua muistia muidenkin käyttöön.

3. Vaatimusten määrittely

Hakumoottorin suunnittelussa lähdettiin alussa liikkeelle määrittelemällä vaatimukset toteutettavan uuden moduulin osalta. Nämä vaatimukset jaoteltiin karkeasti toiminnallisuuteen, laitteistoteknisiin, suorituskykyyn ja käytettävyyteen liittyviin.

3.1 Toiminnalliset vaatimukset

Hakumoottorin on tarjottava hyvin suunniteltu kutsurajapinta ylemmän tason moduuleille. Siinä on määriteltävä ainakin seuraavat toiminnot.

- Moduulin perustiedot
- Moduulin käytön alustus ja lopetus
- Tiedoston puhtauden tutkiminen
- Päätietokantojen päivitystoiminnot
- Sisäisten asetusten muuttaminen ja arvojen kysely
- Inkrementaalisten tietokantojen päivitystoiminnot
- Kutsujan pitää toteuttaa ns. CallBack-toiminnallisuus, jota moduuli hyödyntää raportoidessansa mahdolliset virhetilanteet (tiedosto, muisti tms.), löytämänsä infektiot ja saastuneet tiedostot sekä keskeytystoiminnon, jolla kerrotaan haluttavan lopetettavan senhetkisen tiedoston tutkiminen.

Ensimmäisen kehitysversion hakumoottorin ei tarvitse itse osata puhdistaa virusten tartuntoja. Riittävää on tehokas tunnistaminen, jotta saastuneet tiedostot voidaan asettaa karanteenin odottamaan joko puhdistamista muulla tavoin tai tiedoston kokonaan tuhoamista pois.

Hakumoottori tullaan toteuttamaan dynaamisena, polymorfisena linkityskirjastona, koska näin sen päivittäminen erillisenä moduulina helpottuu. Hakumoottorin virhe- ja infektioraportointien tulee olla muutettavissa paikalliselle kielelle mahdollisimman helposti.

Hakumoottorimoduulin tulee ehdottomasti tukea päätietokannan tietojen päivittämistä inkrementaalisilla päivitystiedoilla, sillä langattomassa ympäristössä siirrettävän tiedonmäärä tulee minimoida. Tarkoituksena on, että inkrementaaliset tietokanta-päivitykset mahtuvat jopa yhteen SMS-viestiin. Näin ollen, koska yhden SMS-viestin pituus on rajattu 160 7-bittiseen merkkiin on käytettävissä 140 tavua tiedonsiirtoon. Tietoturvasyistä päivitysten tulee kuitenkin olla allekirjoitettuja ja tähän toimenpiteeseen varataan 40 tavua. Tästä syystä itse päivitysinformaatioon on

varsinaisesti käytettävissä alle 100 tavua, kun huomioidaan tarvittavat otsikko-, versio-, päiväys-, järjestysnumero ym. tiedot päivityspaketin alussa.

3.2 Laitteistotekniset vaatimukset

Kohdelaitteen prosessorina on ARM-perheen prosessori ja tarkemmin sanottuna *strongArm*-prosessori [13, 25], joka toimii vain 66 MHz:n kellotaajuudella ja näin ollen tämä seikka asettaa tietynlaisia vaatimuksia käsiteltävälle tiedolle. Koska kyseessä on RISC-prosessori on luonnollista, että käsiteltävälle tiedolle on oletuksena sen sijainti oikein asetetulla sanarajalla (word alignment). Toteutuskoodi on oltava ensisijaisesti toimiva nk. little-endian–tavujärjestyksellä, mutta mielellään olisi hyvä toteuttaa koodi niin, että tulevaisuudessa sen kantaminen (porting) olisi mahdollisimman helppoa myös nk. big-endian–tavujärjestystä käyttäville prosessoreille. Itse asiassa ARM-prosessorin käyttäminen on mahdollista sekä big- että little-endian- tilassa ja asia on määriteltävissä prosessorin tilarekistereissä [14]. Tosin nykyisten Symbian-laitteiden oletuksena on lähes poikkeuksetta little-endian, koska tällä tavoin helpotetaan suuresti tiedonvaihtoa PC-koneiden kesken, sillä Intelin suuret prosessoriperheet x86, Pentiumit ovat kaikki little-endian-prosessoriarkkitehtuuria.

Anti-Virus–hakumoottorille asetetaan vaatimuksia myös muistinkäytön suhteen sekä staattiselle että dynaamiselle. Ohjelmakoodi pitää saada mahdollisimman pieneen tilaan, joten toteutustyyli on valittava huolella. Symbian–alustalla on oltava erityisten tarkkana prosessoripinon käytön suhteen, sillä yhteistä säikeen prosessoripinoa on tavallisesti maksimissana käytössä vain noin 12 kB [30]. Siis mitään suuria automaattisia tietorakenteita ei voi varata pinosta edes väliaikaisesti, koska pinoa täytyy olla riittävästi vapaana myös käyttöjärjestelmäkutsuja suoritettaessa. Tästä seuraa lisäksi se tosiasia, ettei myöskään rekursiivinen ohjelmointityyli tule missään nimessä kysymykseen.

Lisäksi ohjelmakoodissa on kaikkialla varauduttava mahdolliseen dynaamisen muistin loppumiseen, koska vielä tämän päivän kämmenmikrolaitteissa ei ole yltäkyllin muistia käytettävissä. Kaikki muistinvarausten epäonnistumiset on tarkistettava välittömästi ja muistin loppuessa pitää välittömästi olla valmis vapauttamaan muistia muidenkin käyttöön, mikäli se vain on mahdollista. Muistia ei saa myöskään pitää turhaan varattuna, koska se on varmaa, että muistia on vähän käytettävissä.

Ohjelmiston vaatimaa muistin määrää mietittäessä tulee ottaa huomioon se tilanne, kuinka paljon muistia laitteessa on normaaliolosuhteissa. Usein tilanne on sellainen, että

vapaata muistia ohjelmien suorittamiseen on vain muutama megatavu, sillä tavallisesti ihmiset pyrkivät asentamaan laitteeseen pieniä apuohjelmia ja tietysti pelejä niin paljon kuin vain on mahdollista. Yleensä PDA- ja kämmenmikrolaitteissa pyörii yhtäaikaisesti useita sovelluksia, jopa yli viisikin erilaista sovellusta, jotka ovat aktiivisessa käytössä. Tällaisia ovat esim. kalenteri, muistiot tapaamisineen, sähköposti ja laskin vain muutamia näistä mainittuna. Verrattuna tavalliseen työasemaan nämä sovellukset saattavat olla päällä yhtämittaisesti useita kuukausia, kun taas usein työasema saatetaan käynnistää uudelleen montakin kertaa työviikon aikana.

Tätä kautta asetetaan hakumoottorille seuraavanlaisia rajoja muistin käytön suhteen. Ohjelmabinäärin koko tulee olla alle 50 kB, pinoa saa käyttää maksimissansa 256 tavua ja dynaamisen muistin varaamisen suhteen tulee pyrkiä alle 64 kB:n jatkuvaan varattuna pitämiseen kuitenkin ehdottomasti niin, että se olisi keskimäärin alle 128 kB, ja myös hetkittäisen varaamisen piikki/huippu tulee olla alle 256 kB.

Symbian–alusta asettaa dynaamisille linkityskirjastototeutuksille lisäksi sellaisen rajoituksen, että kaikki globaali staattinen tieto voi olla vain vakiotietoa ts. sitä ei voi muuttaa ajon aikana.

Hakumoottorin toteutus on testattava huolellisesti niin, ettei missään normaalitilanteessa ole mahdollista saada kohdelaitetta niin jumiin ohjelmakoodissa olevan virheen takia, jotta laite jouduttaisiin käynnistämään uudelleen resetoimalla.

3.3 Suorituskykyvaatimukset

Anti-Virus–hakumoottorin tulee toimia niin, ettei yksittäisen tiedoston tutkiminen saa kestää liian kauan eikä missään nimessä tutkimien saa aiheuttaa laitteen jäämistä jumiin liian suuren tiedoston vuoksi, vaan tarvittaessa on turvauduttava aikarajoittimien laukeamiseen toisin sanoen yksittäistä tiedostoa kohti annetaan maksimiaika, joka on käytettävissä tiedoston puhtauden tutkimiseksi. Mikäli tuon määräajan täyttyessä tiedosto on edelleenkin puhdas eikä siinä ole havaittu yhtään tartuntaa, pidetään tiedostoa puhtaana kokonaisuudessansa, vaikka itse asiassa tiedosto saattaa olla niin suuri tai kompleksinen, että sen tutkimiseen pitäisi käyttää enemmän aikaa.

Mikäli hakumoottori tunnistaa yhdenkin tartunnan eli infektion tiedostossa on tiedoston tutkimiseen lupa käyttää enemmän aikaa, mutta kuitenkin niin, ettei sen tutkiminen vie

kymmentä kertaa kauemmin, kuin mitä vastaavasti muutoin on annettu tiedoston tutkimiseksi aikaa.

Täsmäysalgoritmi on valittava niin, ettei uuden tunnistustiedon lisääminen tietokantoihin hidastuta hakemista ja tiedostojen tutkimista kohtuuttomasti entiseen verrattuna. Ihanteellista olisi ainakin päästä lähelle O(log n)–aikavaativuutta kaikkien virustunnistustietojen lukumäärän suhteen, mutta tätä asiaa ei tutkita sen teoreettisemmin tämän diplomityön aihealueessa, vaan tehdään käytännön havaintoja testiajojen pohjalta. Lopputuloksissa havaitaan, kuinka tunnistetietojen eli sormenjälkien lukumäärä vaikuttaa tiedostojen tutkimisnopeuteen aivan tavallisissa tilanteissa.

3.4 Käytettävyysvaatimukset

Hakumoottori ei saa kuormittaa laitetta niin, että se kuluttaisi turhaan sen akkua tai muuta virtalähdettä. Koska kyseessä on lähes tulkoon sulautetun järjestelmän kanssa toimiminen ei tule myöskään kysymykseen, että hakumoottori varaa itsellensä liikaa keskusyksikköaikaa, vaan toiminta on oltava sellaista, ettei laitteen käytettävyys kärsi liikaa esim. tilanteissa, joissa laitteeseen on tulossa puhelu juuri samanaikaisesti, kun tehdään normaalia tiedostojen tutkimista.

4. Toteutus

Kirjallisuudesta on löydettävissä tutkimustietoa erilaisista merkkijonojen täsmäysalgoritmeista ja niiden ominaisuuksista [9, 15, 21, 31]. Näin ollen voi välttyä siltä, ettei tarvitse keksiä pyörää alusta asti aivan uudelleen ja tietää varmasti, kuinka hidas intuitiivinen nk. *Brute Force* –menetelmä onkaan verrattuna todellisiin tutkittuihin ja kattavasti testattuihin täsmäysalgoritmeihin. Toisin sanoen yritettäisiin soveltaa jotain itse pikaisesti tehtyä täsmäysalgoritmia, jota vieläpä käytettäisiin tutkimiseen tunnistetietojen moninkerroissa jokaiselle hahmolle erikseen. Yksi esimerkki tällaisesta varsin huonosta intuitiivisesta algoritmista on seuraavanlainen, joka on lähes tulkoon tuttu kaikille ennestään [21].

```
void BF(char *y, char *x, int n, int m)
{
  int i, j;

  /* Searching */
  for(i = 0; i <= n - m; i++)
  {
    j = 0;
    while(j < m && y[i + j] == x[j])
      j++;
    if(j >= m)
      OUTPUT(i);
  }
}
```

4.1 Toteutustavan arviointia

Kaikista kriittisimmäksi osaksi toteutustavan valinnassa tulisi olemaan tarkan merkkijonotäsmäysalgoritmin valinnassa ja kuinka löydettäisiin paras algoritmiehdokas. Ei riittäisi ainoastaan, että itse täsmäysalgoritmi on nopea, mutta sen pitäisi mahdollistaa myös tietokantojen mahdollisimman yksinkertaista päivittämistä. Tietokantojen inkrementaaliset päivitykset on oltava niin pieniä kuin mahdollista, jotta ne mahtuisivat annettujen rajojen sisään.

4.1.1 Useamman hahmon samanaikainen tarkka etsintä

Anti-Virus-ohjelmistojen perimmäisenä lähtökohtana on, että tiedostoa on tutkittava kaikkien siihen mahdollisesti tarttuvien virusten suhteen. Tämä tarkoittaa käytännössä sitä, että samanaikaisesti tiedostoa verrataan useampien tuhansien hahmojen

mahdolliseen olemassaoloon haettavassa tiedostossa eli merkkijonossa, jota nimitetään usein yksinkertaisesti tekstiksi. Tämä hankaloittaa ja hidastuttaa tutkimista suuresti.

Ensimmäisenä toteutustapana monen hahmon samanaikaiseen etsintään voidaan luonnollisesti tehdä melko yksinkertaisella tavalla eli soveltaen jotain haluttua täsmäysalgoritmia jokaiselle hahmolle erikseen. Tällöin koko etsintäprosessin aikavaatimus on haettavien hahmojen määrä kertaa algoritmin aikavaatimus. Tämä ei kuitenkaan käytännössä ole järkevin toteutuksen lähtökohta, sillä käytännössä tutkittavien tiedostojen pituuden yhteismäärä on hyvin pian useita gigatavuja ja yleensä ottaen haettaessa hahmoa suurimmaksi pullonkaulaksi nopeuden suhteen muodostuukin tiedostojen käsittelyyn ja lukemiseen kuluva I/O-aika, joka on usein yli 90% koko täsmäysalgoritmin ajankäytöstä.

Perinteisesti samanaikaisesti haettavien hahmojen ongelmaa on lähdetty kirjallisuudessa ratkaisemaan avainpuutekniikoiden avulla (*Keyword Tree* & *Trie*-tekniikoilla). Ehkä tunnetuin tällaisen rinnakkaishaun mahdollistava algoritmi on Ahon ja Corasickin [1] esittelemä algoritmi, joka saadaan myös Knuth-Morris-Pratt-algoritmista [19] muokkaamalla sopivasti sen esittämän korjausfunktion toimintaa. Hyvin usein AC-algoritmin haku on jopa lineaarista, ainoastaan täsmäysten käsittely ja tulostus voi sotkea sen suorituskykyä, erityisesti mikäli hahmot ovat erimittaisia, jolloin alihahmoja pitäisi hakea erikseen. Tässä diplomityön ongelmatapauksessa voidaan kuitenkin rajoittua tarkastelemaan vakiomittaisia hahmoja eli sormenjälkiä, joten varsinaisesti tästä ei näin ollen syntyisi mitään ongelmaa.

Tämä AC-algoritmi voisi olla myös tässä diplomityössä esiteltävän täsmäysalgoritmin lähtökohtana, mutta tarkoituksena on esitellä hieman toisenlainen ratkaisu tähän ongelmaan, joka saattaa jopa olla suorituskyvyltänsä useissa tilanteissa varsin hyvä, jollei toisinaan jopa parempi, mutta silti kuluttaa vähemmän muistia, mikä on tässä tilanteessa jopa nopeutta tärkeämpi ominaisuus.

Yksittäisen hahmon täsmäyksessä on jo melko kauan tiedetty olevan mahdollista tehdä haku pohjautuen hahmon pituudesta riippuviin hyppyihin, jolloin tekstin kaikkia merkkejä ei edes tarvitse tutkia ollenkaan. Tämä tunnetaankin yleisesti Boyer-Moore-algoritmina [5]. Ongelmana onkin vain ollut, onko tämä tekniikka helposti hyödynnettävissä useamman samanaikaisen hahmon täsmäyksessä. Juuri tätä BM-algoritmin esittämää ideaa on tarkoitus soveltaa tässä diplomityössä useamman yhtäaikaisen

hahmon täsmäykselle, jolloin AC-algoritmin lineaarinen aikavaatimus olisi helposti ohitettavissa jopa parempana ratkaisuna.

Suurin hankaluus AC-algoritmin hyödyntämisessä olisi juuri sen esittämän avainpuun vaatima suuri muistimäärä [6] ja tällöin vaatimusmäärittelyjä muistinkulutuksen suhteen ei oikein pystyttäisi täyttämään millään. Itse asiassa avainpuun koko voi hyvin nopeasti kasvaa sellaiseksi, ettei se edes mahtuisi kämmenmikron muistiin ollenkaan, sillä samanaikaisia hahmoja saattaa olla kymmeniä tuhansia, jollei jopa enemmän. Lisäksi AC-algoritmin tapauksessa inkrementaalisten tietokantapäivitysten toteuttaminen olisi käytännössä hyvinkin vaikeaa.

Commentz-Walter [7] esitti algoritmin, joka yhdistää BM-algoritmin tekniikan AC-algoritmiin, ja näin ollen tarjoaa yhden hyvän ratkaisun useamman yhtäaikaisen hahmon täsmäykselle [33]. Kuten edellä jo mainittiin AC-algoritmin avainpuun muistinkulutus on vain liian suuri tämän diplomityön esittämän ongelman ratkaisemiseksi riittävän tehokkaasti.

4.2 Toteutuksen määrittely

4.2.1 Aikaisemmat toteutukset

Tavallisimmin monet aikaisemmat Anti-Virus-hakumoottorit ovat perimmiltänsä pohjautuneet Rabin-Karp-algoritmin [23] edustaman tekniikan hyödyntämiseen. Valitettavasti useimmat tähänastisista hakumoottoreista ovat liian raskaita toteutuksia siirrettäväksi suoraan sellaisenaan kämmenmikrolaitteistoille, joten lienee ainoa oikea ratkaisu on tehdä näille laitteille aivan omansa.

Perinteisesti Anti-Virus-hakumoottorit ovat kiinnostuneita löytämään ensimmäisen jonkun hahmon täsmäyksen, jolloin on siis saatu tunnistus jollekin tunnetulle tietokone-virukselle, madolle tai muulle troijalaiselle. Jos ja kun ensimmäinen täsmäys löytyy, tarkoittaa tämä sitä, että ko. tiedosto pitää saada ensiksi puhdistettua ts. sille pitää suorittaa erikseen määritelty disinfektointitoimenpide, joka voi olla erikseen toteutettu ohjelman osa, erityinen skripti- tai tavukieli tms. Kun tiedosto on saatu puhdistettua aletaan tutkia tiedostoa uudelleen alusta, ettei se vaan sattunut olemaan saastunut useampaan kertaan rekursiivisesti, esim. jonkun hankalan polymorfisen viruksen takia. Vasta kun tiedosto on skannattu kertaalleen kokonaan läpi eikä löydetä enää yhtään täsmäystä, voidaan senhetkisten tietokantatietojen pohjalta pitää tiedostoa puhtaana ja

siirtyä tutkimaan seuraavaa tiedostoa. Uusien virusten löytämisen suhteen auttaakin vain tietokantojen pitäminen ajantasalla niin hyvin kuin mahdollista, jollei AV-hakumoottoriin ole toteutettu tehokasta heuristista analyysiä tiedoston sisällön ja sen suoritustavan emuloinnin suhteen.

4.2.2 Oma ratkaisu

Valittaessa pohjaa täsmäysalgoritmin lähtökohdaksi tutkittavana oli myös, miten voisi hyödyntää likimääräistä merkkijonotäsmäystekniikoita [27]. Valitettavasti projektin puitteissa ei tarjottu aikaa tutkia asiaa sen enempää tieteellisestä näkökulmasta, vaan jotain toimivaa oli saatava mahdollisimman pikaisesti toteutettua ensimmäisiin prototyyppeihin, jolloin myös valittava teknologia tulisi sitomaan toteutuksen tietylle tasolle, mikäli halutaan säilyttää yhteensopivuutta tietokantojen kesken aikaisempiinkin versioihin.

Helpoiten ongelman ratkaisemiseksi pääsisi liikkeelle ottaen lähtökohdaksi Rabin-Karp-algoritmin, jota yritettäisiin muuttaa tarvittaessa paremmaksi. Lisäksi Rabin-Karp-algoritmin valintaa puolustaa se seikka, että sitä hyödyntäen olisi helppo täyttää se kriteeri, ettei uuden tunnistetiedon lisääminen tietokantaan saa hidastuttaa hakua liikaa entiseen verrattuna.

4.3 Toteutus

4.3.1 Rabin-Karp

Hajautus tarjoa helpon tavan välttää turhia merkkivertailuja käytännön tilanteissa. Rabin-Karp-algoritmin ideana on pyrkiä laskemaan uusi hajautusarvo edellisen hajautusarvon ja uuden lisämerkin pohjalta H(s[i..i+m–1]) = hash(H(s[i–1..i+m–2]), s[i+m–1]). Hajautusfunktiolla tulisi olla myös ominaisuutena yhteentörmäysten harvinaisuus. Hajautusfunktiolla saadaan alimerkkijonot muutettua kokonaisluvuiksi, joiden pohjalta on nopeampi analysoida, jollain hitaammalla menetelmällä, voiko ko. tekstin kohta olla halutun hahmon täsmäys. [23]

Jotta RK-algoritmi toimisi nopeasti, olisi hajautusfunktio ja -arvot valittava niin, että ne kukin mahtuvat optimaalisesti prosessorin sananpituuteen eli toisin sanoen useimmiten nykyisillä prosessoreilla 32 bittiin. RK-algoritmin avulla on helppoa toteuttaa sormenjälki- eli signatuuripohjainen etsintä.

4.3.2 Täsmäysalgoritmin perusidea kämmenmikrojen AV-ohjelmistolle

Esitellään ensin täsmäysalgoritmin perusidea, joka pohjautuu Rabin-Karp-algoritmiin. Perusajatuksena on etsiä 8 tavun mittaisia hahmoja nk. sormenjälkiä, joiden pohjalta tapahtuu varsinainen tarkempi tunnistus.

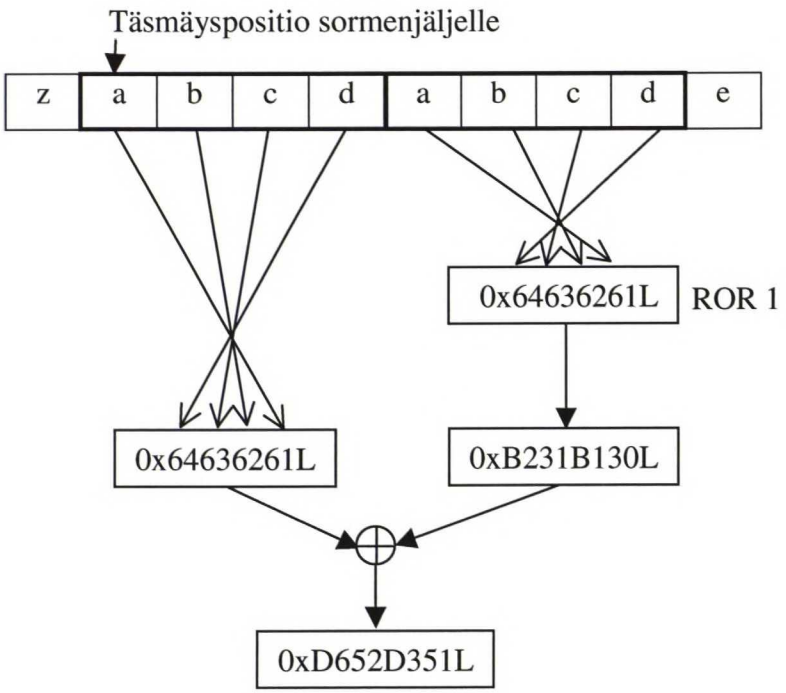
Rabin-Karp-perusalgoritmia on muokattu niin, että yhdistämällä se hajautusarvojen binäärihakuun voidaan samanaikaisesti hakea useampaa hahmoa. Sormenjäljistä, jotka toimivat hahmoina, muodostetaan hajautusarvoja 32-bittisinä lukuina. Nämä luvut tallennetaan kasvavaan järjestykseen taulukkoon, jota hyödynnetään sitten binäärihaun avulla algoritmin loppuvaiheessa, ennen kuin varsinainen täsmäys on mahdollista.

Hajautusarvojen laskeminen on pyritty tekemään mahdollisimman nopeaksi eikä pyrkimyksenä olekaan ollut päästä lähelle täydellisempää hajautusta. Näin ollen hajautusarvon laskeminen perustuu perustoimituksiin kuten xor ja lukujen shiftaaminen tai mikäli prosessori tukee suoraan bittien pyörittämistä eli rotaatiota, hyödynnetään luonnollisesti suoraan sitä toiminnallisuutta.

Hajautusarvojen laskeminen tapahtuu seuraavasti:

```
uint32 hash = AEP32(&hashBfr[i+0]) ^ _lrotr(AEP32(&hashBfr[i+4]), 1);
```

Ja sama esitettynä havainnollisemmin kaavakuvana esimerkkitekstille:



Sormenjäljen eli hahmon 4 ensimmäisestä merkistä muodostetaan little-endian-muotoinen 32-bittinen kaksoissana (*Double word*) ja 4 jälkimmäisestä merkistä vastaavasti oma kaksoissanansa, jolle tehdään bittitason pyöritystä suorittamalla yhden askeleen rotaatio oikealle, jolloin shiftaamisen seurauksena alivuotava bitti pyörähtää takaisin merkitsevimmäksi bitiksi. Rotaatio tehdään, jotta hajautusarvoista tulisi hieman parempia, mutta sen käyttäminen ei ole välttämättömyys ja se voidaan jättää pois muilla prosessoriarkkitehtuureilla havaittaessa lukujen rotatoinnin tai shiftaamisen olevan liian hidasta verrattuna sillä saavutettavaan hajautusarvojen tasaisempaan jakaumaan. Saadut kaksoissanat yksinkertaisesti yhdistetään xor-operaatiolla ja näin saadaan hajautusarvo ko. hahmolle. Hajautusarvon laskeminen on siis huomattavasti yksinkertaistetumpi Rabin-Karpin esittämästä alkuperäisestä ideasta, koska tässä tapauksessa hajautusarvo saadaan kahden yksinkertaisen laskuoperaation tuloksena, kun taas alkuperäisessä RK-algoritmissa käytetään modulo-aritmetiikkaa, joka usein on melko hidasta vielä nykyisilläkin prosessoreilla.

Hajautusarvon laskemisessa suoraan edellä kuvatulla tavalla ei varsinaisesti ole mitään suurta uutta ja lähes samankaltaisia tapoja on käytetty myös muualla pitkään ja hyvällä menestyksellä. Kuitenkin käytettäessä hajautusarvon laskentatapaa, jossa ei ole mukana esimerkiksi bittien rotatointia, vaan molemmat kaksoissanat yhdistetään suoraan yksinkertaisesti xor-operaatiolla, saattaa suorituskyky toisinaan muodostua hieman ongelmalliseksi, mikäli molemmat peräkkäiset kaksoissanat ovat samat, jolloin hajautusarvon tuloksena on nolla. Tämä tietysti riippuu täysin haettavista sormenjäljistä.

4.3.3 Perustäsmäysalgoritmi pseudokoodina

```
Rotate-Bits-Right(value, shift)
  return Truncate-To-32bits((value >> shift) + (value << 32-shift))

Rotate-Bits-Left(value, shift)
  return Truncate-To-32bits((value << shift) + (value >> 32-shift))

// pre-compute the sorted hash-value-table for fixed size patterns length = 8
// P is the multiple pattern table and d number of them
Initialize-Pattern-Table(P, d)
for i = 1 to d
  hash = (P[i][7] << 24) + (P[i][6] << 16) + (P[i][5] << 8) + P[i][4]
  hash = Rotate-Bits-Right(hash, 1)
  hash = hash xor
    ((P[i][3] << 24) + (P[i][2] << 16) + (P[i][1] << 8) + P[i][0])
  sorted-hash-table[i] = hash
```

```
sort-ascending(sorted-hash-table)
retun sorted-hash-table

// T is the text where the multiple patterns are searched
// SHT is the above sorted-hash-table
Rabin-Karp-Matcher(T, SHT)
  n = length(T)
  if n < 8
    return "not found, too short text"
  hash = (T[7] << 24) + (T[6] << 16) + (T[5] << 8) + T[4]
  hash = Rotate-Bits-Right(hash, 1)
  hash = hash xor
    ((T[3] << 24) + (T[2] << 16) + (T[1] << 8) + T[0])
  hash = Rotate-Bits-Left(hash, 9)
  hash = hash xor T[7]

  for i = 0 to n-8
    hash = hash xor T[i+7]
    hash = Rotate-Bits-Right(hash, 9)
    if(Search-For-This-Hash-in-Table(hash, HST) == found)
      then if(P[0..7] == T[i+0..i+])
        then "Pattern " P[0..7] " occurs with shift " i
        else "spurious hit"
    hash = hash xor T[i+0]  // drop lowest, the 1st byte away
    hash = hash xor T[i+4]  // copy the 5th byte to the 4th
    hash = Rotate-Bits-Left(hash, 1)
    hash = hash xor T[i+4]  // drop the duplicate 5th byte away
```

Ei ole aivan itsestään selvää, kuinka uusi hajautusarvo saadaan laskettua vanhasta hajautusarvosta lisäämällä siihen vain yksi uusi tavu. Otetaan esimerkiksi merkkijono ”abcdefghi”, josta ensin poimitaan alusta alimerkkijono ”abcdefgh” ja lasketaan tälle hajautusarvo, joka on 0x64636261L ^ (0x68676665L ROR 1) = 0x64636261L ^ 0xB433B332L = 0xD050D153L. Haluamme edetä eteenpäin niin, että uusi hajautusarvo on laskettu seuraavalle alimerkkijonolle ”bcdefghi”. Siis merkki ”a” tippuu pois yksinkertaisella xor-operaatiolla. Nyt keskeneräinen hajautusarvo on 0xD050D153L ^ 0x61 = 0xD050D132L. Merkki ”e” oli vanhassa hajautusarvossa ylempien 4 tavun joukossa, jolle on tehty yhden askeleen rotaatio oikealle. Nyt merkki ”e” pitää saada siirrettyä kohdasta 4 kohtaan 3. Kohtaan 3. kopiointi voidaan suorittaa jo nyt, ennen kuin hajautusarvolle tehdään uusia rotaatioita. Nyt hajautusarvoksi saadaan yksinkertaisen xor-operaation jälkeen 0xD050D132L ^ 0x65 = 0xD050D157L. Kohdasta 4 poistaminen onnistuu vastaa 1 askeleen rotaatiolla vasemmalle. Siis

(0xD050D157L ROL 1) ^ 0x65 = 0xA0A1A2AFL ^ 0x65 = 0xA0A1A2CAL. Nyt tähän arvoon voidaan lisätä tuleva uusi lisätavu, merkki "i" ja se tapahtuu myös xor-operaatiolla: 0xA0A1A2CAL ^ 0x69 = 0xA0A1A2A3L. Sitten vielä 9 askeleen rotaatio oikealle ja saamme uuden hajautusarvon halutulle merkkijonolle "bcdefghi". Siis tämä tapahtuu näin (0xA0A1A2A3L ROR 9) = 0x51D050D1L, joka on sama tulos kuin laskemalla suoraan hajautusarvo merkkijonolle "bcdefghi" eli 0x65646362L ^ (0x69686766L ROR 1) = 0x65646362L ^ 34B433B3L = 0x51D050D1L. Kaiken kaikkiaan tähän kuluu vain noin 10 yksinkertaista konekielistä käskyä riippuen hieman prosessoriarkkitehtuurista.

Edellä kuvattu operaatio ei välttämättä ole aivan helppo ymmärtää, mutta toki operaatiot on mahdollista suorittaa myös aivan toisessa järjestyksessä, jolloin edellä olevan paikkansa pitävyys olisi ehkäpä helpompi havaita. Kyseessä on Boolen algebran xor-operaation kommutatiivisuus. Tätä ei kuitenkaan todisteta tässä diplomityössä, koska tämä on vain esimerkkinä, kuinka uusi hajautusarvo voidaan laskea vanhasta lisäämällä siihen vain yksi uusi tavu, poistamalla alin tavu sekä siirtämällä 4. tavu 3. tavuksi. Tällä tavoin toimien voidaan välttää sitä ongelmatilannetta, joka yleensä syntyy RISC-prosessoreilla esim. ARM-perheen prosessoreilla, sillä ne eivät salli 32-bittisten kaksoissanojen lukemista muista kuin neljällä jaollisista osoitteista tai muuten seurauksena on hankalia poikkeustilanteita (alignment exception). Toisilla prosessoriarkkitehtuureilla, kuten esim. Intel, on aivan sama, vaikka kaksoissanoja luetaan mistä tahansa osoitteesta, koska seuraus on vain, että muutamia ylimääräisiä kellojaksoja kuluu dataa luettaessa ei-neljällä jaollisista osoitteista. Intelin prosessoreilla onkin nopeampaa lukea molemmat kaksoissanat uudelleen seuraavassa positiossa ja laskea näiden pohjalta seuraava hajautusarvo kuin, että laskea uusi hajautusarvo vanhasta edellä kuvatulla tekniikalla. Tällöin yksinkertaisia käskyjä tarvitaan neljä ja huomioiden mahdolliset ylimääräiset kellojaksot, vastaa tilanne 4-7 käskyä. Niiden prosessoreiden kohdalla, jotka vaativat lukuja oikein jaollisista osoitteista tai olettavat big-endian-tavujärjestystä, on edellä kuvattu laskentapa vanhasta hajautusarvosta nopeampi, koska muutoin toimimalla tarvitaan yli 20 käskyä, joista lähes joka toinen tavujen shiftaamista.

4.3.4 Perustäsmäysalgoritmi toteutettuna C++:lla

```
#ifdef NEED_ROTATE
//Unless your processor supports better rotating instructions
static uint32 _lrotl(uint32 x, int i)
```

```
{
  i &= 0x1F;     // make also negative rotations possible
  if(i)
    x = ((x << i) | (x >> (0x20 - i)));
  return x;
}
static uint32 _lrotr(uint32 x, int i)
{
  i &= 0x1F;     // make also negative rotations possible
  if(i)
    x = ((x >> i) | (x << (0x20 - i)));
  return x;
}
#endif  // NEED_ROTATE

#if ALIGN_DATA || ENDIAN
#define HASH_METHOD 1
//Define your AEP32 macro to make 32 bit access to possibly unaligned address
//Aligned Endian Pointer 32, replace with a more efficient one ;-)
#define AEP32(x) \
  ((x)[0]+((uint32)(x)[1]<<8)+((uint32)(x)[2]<<16)+((uint32)(x)[3]<<24))
#else   // ALIGN_DATA || ENDIAN
//Lucky you, maybe you have an Intel processor
#define AEP32(x) *x
#define HASH_METHOD 0
#endif  // ALIGN_DATA || ENDIAN

// if HASH_METHOD == 1 unaligned double word reads are allowed at any address

bool matchHashes(const uint8 *const hashBfr, int i, const int j)
{                                   // buffer, start pos, end pos
#if HASH_METHOD
  uint32 hash = _lrotl(AEP32(&hashBfr[i+0]) ^
    _lrotr(AEP32(&hashBfr[i+4]), 1), 9) ^ hashBfr[i+7];
#endif  // HASH_METHOD
  do {
#if HASH_METHOD
    hash = _lrotr(hash ^ hashBfr[i+7], 9);
#else   // HASH_METHOD
    uint32 hash = AEP32(&hashBfr[i+0]) ^ _lrotr(AEP32(&hashBfr[i+4]), 1);
#endif  // HASH_METHOD
    const uint32 *top = hashTable32hi;
    const uint32 *mid = hashTable32lo-1;
    while(top > mid)    // unless the hash was not in the database
    {                   // do the good old binary search
      uint32 k = (top - ++mid) >> 1;
      uint32 h = *(mid += k);
```

```
        if(h == hash)
        {
          do {
            --mid;                    // scan for hash duplicates in the table
          } while(*mid == hash);      // and start with the first one
          k = (mid++ - lo) << 3;
          k += seekPosInfoTable;      // add to 8 byte extra info file position
          do {
            if(identify(i, k+=8, hash))
              return true;            // the first exact detection is enough
            mid++;
          } while(*mid == hash);
          break;
        }
        else if(h > hash)
        {
          top = --mid;
          mid -= k;
        }
      }
#if HASH_METHOD
// Rehash it
      hash = _lrotl(hash ^ hashBfr[i+0] ^ hashBfr[i+4], 1) ^ hashBfr[i+4];
#endif  // HASH_METHOD
      ++i;
    } while(j > i);
    return false;
}
#undef HASH_METHOD
```

4.4 Täsmäysalgoritmin muokkaamista edelleen paremmaksi

Edellä esitetty algoritmi on hyvä lähtökohta ongelman ratkaisuun, mutta voisiko sitä vielä muuttaa paremmaksi? Tunnetusti käytännössä kaikkein tehokkaimmat ratkaisut yhden hahmon etsintään ovat pohjautuneet Boyer-Moore-algoritmiin ja sen johdannaisiin [28]. Ei ole kuitenkaan mitenkään varmaa, olisiko BM-algoritmin perusideat sellaisenaan sovellettavissa useiden samanaikaisten hahmojen etsintään, varsinkin kun tilanne on pian se, että tulevaisuudessa tietokannassa olevia hahmoja tulee olemaan kymmeniä tuhansia.

Seuraavana esitettävät algoritmin tehostamistavat tarvitsevat etukäteislaskentaa vain kerran jokaista koko tietokannan sisältöä varten ja tarvittaessa tieto voidaan jopa sisällyttää tietokantaan valmiiksi laskettuna. Ainoan ongelman saattaa muodostaa

tietokannan inkrementaaliset päivitystilanteet, mutta näihinkin ongelmiin on löydettävissä ratkaisut melko helposti.

4.4.1 Kaksitasoinen hajautus

Ensimmäinen parannuskohta tehokkuuden suhteen voisi olla tapa, jolla voidaan välttää turhan binäärihaun käyttöä. Tähän on olemassa hyvin nopea ratkaisu kaksitasoisen hajautuksen kautta, mutta sen hyödyntämiseksi joudutaan käyttämään lisää muistia. Kuitenkin tarvittava lisämuistin määrä on melko pieni ja itse asiassa jo 8 kilotavua on aivan riittävä tässä tilanteessa.

Koska hajautusarvomme ovat 32-bittisiä lukuja, muodostamme näiden pohjalta puolet pienemmät uudet hajautusarvot eli 16-bittiset luvut. Kiinnostavinta tässä vaiheessa on vain tietää, onko mahdollista, että kyseinen hajautusarvo voisi löytyä binäärihaun avulla varsinaisesta kasvavaan järjestykseen lajitellusta hajautusarvotaulusta. Näin ollen 16-bittiset hajautusarvot voidaan tallentaa yksinkertaiseen bittitaulukkoon, jonka kooksi muodostuu 2^{16} / 8 eli 8192 tavua.

Hieman vastaavanlaisen kaksitasoisen hajautustekniikan ovat esittäneet jo aikaisemmin Robert Muth ja Udi Manber, mutta tästä huolimatta tässä diplomityössä esitetty tapa ja heidän esittämänsä poikkeavat melko paljonkin toisistansa [22]. Heidän tapansa kuluttaa hieman enemmän muistia viimeisen hajautuksen muodossa, kun taas tässä esitetyssä tavassa viimeinen haku tapahtuu aina binäärihaun avulla.

Toisen tason hajautustaulun arvot lasketaan myös yksinkertaisella ja nopealla tavalla:

```
uint16 hash16 = (uint16) ((hash32 >> 16) ^ hash32);
```

4.4.2 Boyer-Moore-algoritmin hyödyntäminen

Boyer-Moore-algoritmin alkuperäisen idea on lyhykäisyydessänsä seuraava [9]. Hahmoa siirretään vasemmalta oikealle ja samalla jokaisessa pysähdyskohdassa eli täsmäysoperaation alussa merkkien vertaileminen aloitetaan hahmon lopusta ja edetään tarvittaessa takaisinpäin siis toisin sanoen vasemmalle. Mikäli vertailtavat merkit eivät täsmää keskenänsä, käytetään etukäteen taulukkoon valmiiksi laskettua siirtofunktiota hahmon siirtämiseksi 1..m askelta oikealle. Parhaimmillaan algoritmissa tarvitsee tutkia merkkijonoa vain hahmon pituuden moninkerroissa.

4.4.2.1 Ei-täsmänneen viimeisen merkin sääntö

Tämän idea pohjautuu Horspoolin esittämään pieneen muutokseen, joka on hieman yksinkertaistetumpi versio yleisestä BM-algoritmista. Yleisesti se tunnetaankin Boyer-Moore-Horspool-algoritmina [17].

Muodostetaan BMH-algoritmin ideoita hyödyntäen kaikkien haettavien hahmojen pohjalta yksinkertainen taulukko, jossa on aakkoston jokaiselle merkille arvo, esiintyykö ko. merkki jonkin samanaikaisesti haettavan hahmon jossain positiossa vai eikö se ollenkaan esiinny yhdessäkään hahmossa. Arvo 0 tarkoittaa, ettei merkki ole yhdessäkään hahmossa, muutoin kyseinen merkki on jokin hahmon osana.

Tämä algoritmin nopeutuskeino toimii varsinkin alussa hyvin, kun tietokantaan on lisätty vain muutamien hahmojen tunnistamistiedot. Muulloinkin, mikäli hahmoissa esiintyy paljon samoja merkkejä, on tästä säännöstä hyötyä todella paljon. Valitettavasti heti, kun hahmoissa esiintyy kaikki aakkoston merkit, menettää tämä nopeutustapa etunsa. Silloin kannattanee poistaa tämä sääntö pois koko täsmäysalgoritmista. Tilanne saattanee syntyä vasta silloin, kun tietokantaa on lisätty useita satoja sormenjälkien hahmoja.

Lisämuistia tämän parannuksen toiminnallisuuden toteuttamiseksi vaaditaankin tässä tilanteessa aakkoston koko eli vain 256 tavua, mikä on lähes kuin juuri ei mitään tai tehtäessä sama bittitauluna vain 32 tavua.

4.4.2.2 Ei-täsmänneen merkkiparin sääntö

Ei-täsmänneen merkin sääntöä on hyvin helppo laajentaa koskemaan 2 merkin pareja, siinä järjestyksessä kuin ne mahdollisesti esiintyvät hahmoissa. Näin ollen tässä tapauksessa yhdestä hahmosta saadaan tällöin 7 merkkiparia, koska käytämme 8-merkkistä sormenjälkitekniikka. Käydään läpi kaikki tietokannan hahmot ja kerätään kaikki esiintyvät merkkiparit bittitauluun. Lisämuistia tämän tekniikan toteuttaminen vaatii yksinkertaisimmillansa yhtä paljon kuin kaksitasoisen hajautuksen toteuttaminen eli 8 kB.

Mutta viedään ideaa hieman pitemmälle ja muodostetaan sormenjäljistä 2-merkin alimerkkijonotauluja eli puhutaan ns. 2-grammeista. Koska sormenjälkien pituudeksi on valittu 8 merkkiä/tavua jaetaan sormenjäljet 7 eri positioon. 1. position muodostavat 1. ja 2. merkin pari ja viimeisen 7. position 7. ja 8. merkin pari. Kerätään 2-grammien informaatiota kumulatiivisesti niin, että 1. positiosta tallennetaan sama tieto myös 2.-7.

positioiden tiedoksi sekä 2. positiosta 3.-7. positioiden tiedoksi. Näin edetään, kunnes on jäljellä vain viimeisen 7. position tieto, joka ei kumuloidu muualle. Lisämuistia tämän algoritmin parannustapa vaatii 7*8 kB eli 56 kB, mikä ei kuitenkaan ole mitenkään liikaa, koska sillä saavutettavat edut ovat mitä ilmeisintä.

Tehdään tästä esimerkki Eicar-testimerkkijonon kahdeksan ensimmäisen merkin pohjalta.

0x58 0x35 0x4F 0x21 0x50 0x25 0x40 0x41 *X5O!P%@A*

```
uint8 badCharPair[7][8192]; // 2-grams
memset(badCharPair, 0, sizeof(badCharPair));
```

positio 1. — **0x58 0x35** *X5*

```
// 0x3558->0x06AB
for(int i = 0; i < 7; i++)
  badCharPair[i][0x06AB] |= 0x01;
```

positio 2. — **0x35 0x4F** *5O*

```
// 0x4F35->0x09E6
for(int i = 1; i < 7; i++)
  badCharPair[i][0x09E6] |= 0x20;
```

positio 3. — **0x4F 0x21** *O!*

```
// 0x214F->0x0429
for(int i = 2; i < 7; i++)
  badCharPair[i][0x0429] |= 0x80;
```

positio 4. — **0x21 0x50** *!P*

```
// 0x5021->0x0A04
for(int i = 3; i < 7; i++)
  badCharPair[i][0x0A04] |= 0x02;
```

positio 5. — **0x50 0x25** *P%*

```
// 0x2550->0x04AA
for(int i = 4; i < 7; i++)
  badCharPair[i][0x04AA] |= 0x01;
```

positio 6. — **0x25 0x40** %@

```
// 0x4025->0x0804
for(int i = 5; i < 7; i++)
  badCharPair[i][0x0804] |= 0x20;
```

positio 7. — **0x40 0x41** @*A*

```
// 0x4140->0x0828
badCharPair[6][0x0828] |= 0x01;
```

Tämä yllä kuvattu toimenpide suoritetaan kerran kaikille tietokannan sisältämille hahmoille. Tämä edellä esitetty algoritmin nopeutustapa on jo hyvin lähellä Sun Wun tohtorinväitöskirjassansa esittämää tapaa [32]. Tämä Wun algoritmi kuvaillaan tuossa väitöskirjassa sivuilla 47-50. Tässä diplomityössä esiteltävän algoritmin voidaan ajatella vastaavan tilannetta, jossa Wun kuvaamien huonojen lohkojen (*bad blocks, a fixed size bad string*) koko on 2. Itse asiassa Wu kuvaa samantapaista toimenpidettä rakennettaessa *MEMBER*-taulua näille lohkoille esiprosessoinnin aikana ja analogia edellä kuvatulle Ei-täsmänneen merkkipari- eli badCharPair-taulun rakentamiselle on täysin ilmeistä, molemmissa tapauksissa tunnetut lohkot tai merkkiparit merkitään arvolla 1 muutoin tiedetään, että arvon 0 olevia ei ole kohdattu missään hahmon rakenteessa. Juuri edellä olevaa läpikäyntiä merkkiparien positioista 1..7 voidaan ajatella kuvaavan Wun esittelemään ikkunan liukumistekniikkaa, kun tuon ikkunan koko on vastaavasti 2. Pitää kuitenkin huomioida, että Wun algoritmi on suunnattu yksittäisen hahmon etsimiseen sallien likimääräinen täsmäys.

Wun algoritmissa on vain yksi *MEMBER*-taulu, kun tässä diplomityössä esitellyssä algoritmissa on mukana 7 vastaavanlaista bittitaulua, jotka ovat sidottuina yhtäaikaisten hahmojen eli sormenjälkien vastaaviin positioihin merkkiparien muodossa. Wun käyttämä *MEMBER*-taulu vastaa ainoastaan tämän diplomityön algoritmin 7. position badCharPair-taulua, johon on kumuloitunut myös kaikkien edeltävien positioiden merkkiparit. Juuri tämä pieni eroavaisuus tavoissa mahdollistaakin tekniikan hyödyntämisen paremmin useamman hahmon samanaikaiseen etsintään. Wun kuvaama *map*-funktio merkkijonolle, joka on vakiomittainen lohko, on tämän diplomityön tapauksessa hyvin yksinkertainen: map(s) = CharToInt(s[0]) + CharToInt(s[1]) << 8; toisin sanoen tyyppipakotusten kautta little-endian-muodossa: map(s) = (uint16) s; ja tämä, jos mikä, on nopea toimenpide suorittaa.

Wu ja Manber ovat kyllä esittäneet, kuinka edellä kuvattu Wun algoritmi on hyödynnettävissä useamman hahmon samanaikaiseen etsintään [33]. Kuitenkin lopputulos on se, että erovaisuus algoritmien välillä on sama kuin edellisessäkin tapauksessa. He käyttävät kahta *MEMBER*-taulua (*prefix and suffix tables*), joista *suffix*-taulua käytetään tehtäessä BMH-tyylisiä hyppyjä (*SHIFT table*), kun tässä diplomityössä hyödynnetään 7 vastaavaa bittitaulua. Wun ja Manberin yhteinen algoritmi kuitenkin muistuttaa jo hieman enemmän tämän diplomityön esittämää algoritmia. Heidän esittämän 3 merkin mittaisen alimerkkijonotekniikan soveltamista hajautuksen kautta saattaisi kannattaa kokeilla myös tässä esitettyyn algoritmiin.

4.4.3 Uusi paranneltu täsmäysalgoritmi pseudokoodina

Lähdetään kehittämään uutta täsmäysalgoritmia BMH-algoritmin esittämien tekniikoiden pohjalta. Koska hahmoina on sormenjälkiä, joiden oletetaan esiintyvän hyvin harvoin, kannattaa algoritmi toteuttaa siitä näkökulmasta, että pyritään tekemään aina niin paljon BMH-tyylisiä hyppyjä kuin vain on mahdollista ja pyritään suosimaan mahdollisimman pitkien hyppyjen tekemistä. Tätä ideaa kuvataan lyhyesti seuraavaksi esitettävällä pseudokoodilla.

```
Fast-Multi—Pattern-Matcher(T)
  n = length(T)
  for i = 0 to n-8
    if Search-All-Patterns-for-This-Char(T[i+7]) == not-found
      then i += 8
    else if Search-All-Patterns-for-This-CharPair(T[i+6],T[i+7],6) ==
            not-found
      then i += 7
    else if Search-All-Patterns-for-This-CharPair(T[i+5],T[i+6],5) ==
            not-found
      then i += 6
    else if Search-All-Patterns-for-This-CharPair(T[i+4],T[i+5],4) ==
            not-found
      then i += 5
    else if Search-All-Patterns-for-This-CharPair(T[i+3],T[i+4],3) ==
            not-found
      then i += 4
    else if Search-All-Patterns-for-This-CharPair(T[i+2],T[i+7],2) ==
            not-found
      then i += 3
    else if Search-All-Patterns-for-This-CharPair(T[i+1],T[i+2],1) ==
            not-found
      then i += 2
```

```
    else if Search-All-Patterns-for-This-CharPair(T[i+0],T[i+1],0) ==
            not-found
      then i += 1
    else
      do while(i < n-8)
        hash = Compute-Hash(T[0..7])
        if(Second-Level-Hashing(hash) == found)
          then if(Search-For-This-Hash-in-Table(hash, HST) == found)
            then if(P[0..7] == T[i+0..i+])
              then "Pattern " P[0..7] " occurs with shift " i
              else "spurious hit"
        i = i + 1
        if(i >= n-8)
           break
        if Search-All-Patterns-for-This-Char(T[i+7]) == not-found
          then i += 8
               break continue-in-outer-for-loop
        else if Search-All-Patterns-for-This-CharPair(T[i+6],T[i+7],6) ==
                not-found
          then i += 7
               break continue-in-outer-for-loop

Search-All-Patterns-for-This-Char(c)
  // make sure that no pattern can have the c character in any position
  // in this true return not-found otherwise found

Search-All-Patterns-for-This-CharPair(c1, c2, p)
  // make sure that no pattern can have the character pair c1&c2
  // in any pattern and in any pair-positions [0..p]
  // pattern pair-positions:     [  1 ][  3 ][  5 ]
  // pattern positions:       [0][1][2][3][4][5][6][7]
  // pattern pair-positions: [  0 ][  2 ][  4 ][  6 ]
  // in this true return not-found otherwise found
```

4.4.4 Uusi paranneltu täsmäysalgoritmi C++:lla

```
#ifdef NEED_ROTATE
//Unless your processor supports better rotating instructions
static uint32 _lrotl(uint32 x, int i)
{
  i &= 0x1F;     // make also negative rotations possible
  if(i)
    x = ((x << i) | (x >> (0x20 - i)));
  return x;
}
```

```
static uint32 _lrotr(uint32 x, int i)
{
  i &= 0x1F;    // make also negative rotations possible
  if(i)
    x = ((x >> i) | (x << (0x20 - i)));
  return x;
}
#endif  // NEED_ROTATE

//for little-endian processors
#define ENDIAN 0
//for big-endian processors
//#define ENDIAN 1

bool matchHashes(const uint8 *const hashBfr, int i, const int j)
{                                  // buffer, start pos, end pos
  static const uint8 andMask[8] = {
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };
  union {
    uint8  db[8];
    uint32 dw[2];
  } buffer;
  const uint8* pos = hashBfr - 1;
  const uint8* const limit = pos + j;
  pos += i;
  while(pos <= limit)
  {
    uint yy = *(pos += 8);
    if(!badLastChar[yy])
      continue; // Boyer-Moore-Horspool, bad last char rule for any signature
    uint8* tp = &buffer.db[7-ENDIAN*7];
    uint8* x2h = badCharPair + 7*0x2000;
    for(;;)
    {
      *tp = yy;
      uint xx = yy << 5;
      yy = *--pos;      // backing one step and make a comparison
      if(!((x2h-=0x2000)[(yy>>3)+xx]&andMask[yy&7]))    // 7th .. 1st 2-grams
        break;          // BMH, no signature can have this pair of chars
      tp -= 1-2*ENDIAN;
      if(tp != &buffer.db[ENDIAN*7])
        continue;       // not yet at the start of potential match
      *tp = yy;         // now we have a very potential match
      uint32 hash = _lrotr(buffer.dw[1-ENDIAN], 1) ^ buffer.dw[0+ENDIAN];
      for(;;)
      {
        uint32 const b = hash ^ (hash >> 16);          // 2nd level hashing
```

```
        if(hash2ndLevel[((b>>3)&0x1FFF)]&andMask[b&7])  // a good candidate?
        {
          const uint32* top = hashTable32hi;
          const uint32* mid = hashTable32lo-1;
          while(top > mid)       // unless the hash was not in the database
          {                      // do the good old binary search
            uint32 const k = (top - ++mid) >> 1;
            uint32 const h = *(mid += k);
            if(h == hash)
              if(identifyThis(hash, mid, pos))  // scan for duplicates etc.
                return true;
              else
                break;
            if(h > hash)
            {
              top = --mid;
              mid -= k;
            }
          }
        }
        // that hash wasn't in the database
        uint zz=*(pos += 8);
        if(limit < pos) // Remember to check the Stop Condition, too!
          return false;
        if(!badLastChar[zz])     // Could a BMH step be possible?
          break;         // BMH, bad last char rule for any signature, pos+=8
        uint ww = (zz << 5) + 6*0x2000;
        zz = *--pos;
        if(!(badCharPair[(zz>>3)+ww]&andMask[zz&7])) // 7th 2-gram usable?
          break;         // BMH, no signature can have this substring, pos+=7
#if ALIGN_DATA || ENDIAN
        ww = (pos -= 6)[3];      // compute a new hash using the previous one
        hash = _lrotl(hash ^ ww ^ pos[-1], 1);  // as a base for it
        hash = _lrotr(hash ^ ww ^ pos[7], 9);   // the new hash is ready
#else   // ALIGN_DATA || ENDIAN
        pos -= 6;        // It could be much faster this way e.g. with Intel
        hash = _lrotr(*(uint32*)&pos[4], 1) ^ *(uint32*)&pos[0];
#endif  // ALIGN_DATA || ENDIAN
      }
      break;
    }
  }
  return false;
}
```

4.4.5 Uuden täsmäysalgoritmin analysointia

Nopeasti analysoimalla edellä esitettyä uutta täsmäysalgoritmia voidaan havaita seuraavaa. Hahmojen täsmääminen jossain tietyssä positiossa alkaa tutkimalla tuon position 8. merkkiä ja etenemällä sen pohjalta joko 8-askeleen BMH-tyylisellä hypyllä eteenpäin tai tutkien saman position 7. ja 8. merkin muodostavaa merkkiparia, jollei 8-askeleen hyppy ole suoraan mahdollinen. Useimmiten täsmäyksen aikana verrataan vain joka 8. tavua tekstissä eli siis aina, mikäli senhetkinen tutkittava merkki (tekstin position 8. merkki) ei ole ei-täsmäävän viimeisen merkin taulussa eli kyseisen aakkoston merkin arvo on tuossa taulussa 0. Mutta kun sattuu tilanne, jossa 8. merkki on mukana ko. taulussa eli arvo on eri kuin 0, verrataan merkkien 7. ja 8. muodostavaa paria, onko se ei-täsmänneiden merkkiparien taulussa. Aina kun merkkipari ei ole mukana tässä taulussa, voidaan hahmon täsmäystä jatkaa tekstin alkuperäisestä positiosta 7 askeleen siirrolla oikealle, muulloin seuraavaksi edetään peruuttamisessa vasemmalle eli taaksepäin edeten tutkitaan 6. ja 7. muodostavaa merkkiparia, jolloin positiota voitaisiin siirtää 6 askeleella, ja tätä jatketaan aina 1. ja 2. merkin muodostavaan pariin, jolloin on löytynyt mitä todennäköisin 8 merkin muodostama hahmo. Vasta silloin tälle hahmolle lasketaan vihdoinkin sen hajautusarvo ja varmistetaan, että myös kaksitasoisen hajautustaulun mukaan on mahdollista, että hajautusarvo löytyisi binäärihaun avulla. Jollei hajautusarvoa löydy binäärihaun kauttakaan, tutkitaan nopeasti tilanne, olisiko mahdollista suorittaa uusi BMH-tyylinen hyppy joko 8 tai 7 askelta. Jollei se ole mahdollista, lasketaan RK-algoritmin mukaisesti uusi hajautusarvo vanhan pohjalta hyödyntäen uusinta 8. kohdassa olevaa merkkiä ja palataan suoraan tarkastamaan tilannetta kaksitasoinen hajautuksen suhteen. Pyrkimyksenä on päästä tekemään uudelleen BMH-hyppyjä niin pian kuin se vain on mahdollista, muulloin lasketaan uudet hajautusarvot vanhojen pohjalta RK-algoritmiin pohjautuen. Mutta vasta sitten kun hajautusarvo löytyy tietokannan hajautusarvojen taulukosta voidaan sanoa melko varmaksi, että ollaan löydetty erittäin todennäköinen täsmäys ko. hahmolle. Tarkempi analyysi tälle algoritmille löytyy luvussa *5 Tulosten tarkastelu* sivulla 41.

Kannattanee lisäksi kiinnittää huomiota, millainen tässä diplomityössä esitetyssä täsmäysalgoritmissa toteutettu binäärihaku on optimoituna ja millaiseksi se kääntyy assembly-kielitasolla eri prosessoreille. Aikaisemmin tämän diplomityön tekijä ei ole kohdannut yhtä tehokkaaksi toteutettua versiota, joka vaatisi yhtä vähän käskyjä ja on silti samalla optimoitu täysin nopeuden suhteen.

```
const uint32 hash = hashTry;
const uint32* top = hiBound;
const uint32* mid = loBound-1;
while(top > mid)
{               // do the good old binary search
  uint32 const k = (top - ++mid) >> 1;
  uint32 const h = *(mid += k);
  if(h == hash)
    break;      // found it in the table, do also something else if needed!
  if(h > hash)
  {
    top = --mid;
    mid -= k;
  }
}
```

Tämä kääntyy muun muassa seuraavanlaiseksi, Intel x86 – 386 optimoinneilla:

```
; 1    : const uint32 hash = hashTry;
       mov     edx, DWORD PTR _hashTry$[esp]
; 2    : const uint32* top = hiBound;
; 3    : const uint32* mid = loBound-1;
       mov     eax, DWORD PTR _loBound$[esp]
       sub     eax, 4
       mov     ebx, DWORD PTR _hiBound$[esp]
$L1:
; 4    : while(top > mid)
; 5    : {               // do the good old binary search
       cmp     ebx, eax
       jbe     SHORT $L3
; 6    :   uint32 const k = (top - ++mid) >> 1;
       add     eax, 4
       mov     ecx, ebx
       sub     ecx, eax
       sar     ecx, 3
; 7    :   uint32 const h = *(mid += k);
       shl     ecx, 2
       mov     edi, DWORD PTR [eax+ecx]
       add     eax, ecx
; 8    :   if(h == hash)
       cmp     edi, edx
       je      SHORT $L2
; 9    :     break;      // found it in the table, do also something else if needed!
; 10   :   if(h > hash)
       jbe     SHORT $L1
; 11   :   {
; 12   :     top = --mid;
       sub     eax, 4
```

```
        mov     ebx, eax
; 13   :     mid -= k;
        sub     eax, ecx
; 14   :       }
; 15   :     }
        jmp     SHORT $L1
$L2:
; 9    :     break;      // found it in the table, do also something else if needed!
$L3:
```

StrongArm-prosessorilla:

```
 mov    ip, [sp, #8]
 mov    r2, [sp, #16]
 sub    r2, r2, #1
 mov    r0, [sp, #12]
.L1:
 cmp    r0, r2
 bls    .L3
 add    r2, r2, #4
 rsb    r3, r2, r0
 mov    r3, r3, asr #3
 mov    r1, r3, asl #2
 ldr    r3, [r2, r1]!
 cmp    r3, ip
 beq    .L2
 subhi  r2, r2, #4
 movhi  r0, r2
 rsbhi  r2, r1, r2
 b      .L1
.L2:
.L3:
```

Pentium Pro:lla ja sitä paremmilla Intelin prosessoreilla olisi mahdollista kytkeä käyttöön StrongArmin kaltaiset ehdolliset siirtokäskyt (CMovCC-käskyt) ja muut lisäoptimoinnit, jolloin edellä olevat käännökset alkaisivat muistuttaa toisiansa hyvinkin paljon. Binäärihaku on siis mahdollista toteuttaa varsin tehokkaasti optimoituna kohdekoneessa edellä kuvatulla tavalla.

4.4.6 Algoritmin lisäparanteluajatuksia

4.4.6.1 Lopetusehdon tarkkailukohta

Täsmäyspuskurin rajojen vertailu on mahdollista siirtää pois aivan silmukan sisäosasta [`while(pos <= limit)`], jos ja vain jos lisätään ko. puskurin loppuun joku varmasti

olemassa oleva hahmo tai niiden oikeita osia, jotta etsintä päättyisi asianmukaisesti, ja vasta merkkiparien vertailun aikana varmistetaan, ettei vain oikeasti olla jo puskurin lopussa, jolloin emme haluakaan suorittaa täsmäystä sen pidemmälle, sillä haettavaa hahmoa ei todellisuudessa löytynyt. Tämä idea on vastaavanlainen kuin mitä on jo esitetty Boyer-Moore-Horspool-Hume-Sunday-algoritmissa [18, 26]

Tällöin päästäisiin lähes tulkoon tällaiseen toteutukseen

```
for(;;)
{
  uint yy;
  do {        // Boyer-Moore-Horspool, Bad last char rule for any signature
    yy = *(pos += 8);
  } while(!badLastChar[yy]);
```

ja

```
    if(!((x2h-=0x2000)[(yy>>3)+xx]&andMask[yy&7]))     // 7th .. 1st 2-grams
      break;          // BMH, no signature can have this pair of chars
    if(limit < pos)   // Stop Condition Checking moved to here
      return false;
    tp -= 1-2*ENDIAN;
    if(tp != &buffer.db[ENDIAN*7]);
      continue;       // not yet at the start of potential match
```

Tällöin täsmäysalgoritmin suoritusaika kuluisi enimmän aikaa tuossa tiukassa silmukassa tehden ei-täsmäävän viimeisen merkin testiä sekä silloin tällöin ei-täsmäävien merkkiparien esiintymisen tarkastamiseen, jolloin varmistetaan, ettei oltaisi puskurin lopussa, ja tarvittaessa sitä kautta voitaisiin edetä binäärihakuosuuteen asti.

Useimmat nykyaikaiset kääntäjät osaavat tehdä tuolle tiukalle silmukalle optimointia automaattisesti kohdeprosessorin suhteen purkamalla silmukkaa auki (*loop unrolling*) tehden useamman merkin vertailua silmukan sisällä. Mutta mikäli kääntäjä ei tätä jostain syystä tee automaattisesti, voisi kokeilla ideaa joka on esitetty Hume-Sundayn-laajennuksessa BMH-algoritmiin [18] ja heidän tuloksissansa kolmen kierroksen aukipurkamisella saavutettaisiin useimmissa – ainakin vanhemmissa – prosessori-arkkitehtuureissa paras tulos.

4.4.6.2 Muiden algoritmien hyödyntäminen

Nykyisessä algoritmissa on pitäydytty edelleen RK-algoritmin ja binäärihaun muodostamassa tekniikassa lopullisen täsmäyksen suhteen. Mitä ilmeisimmin tämä

osuus voisi olla korvattavissa esim. Shift-Or-algoritmilla nopeamman täsmäys-algoritmin toteuttamiseksi [4, 29]. Tässä diplomityössä tätä ei kuitenkaan ole toteutettu, koska on selvää, että tämä lähestymistapa vaatisi enemmän muistia käyttöönsä ja ainakaan kämmenmikrolaitteistolla tämä ei ehkä olisi se järkevin toteutusvaihtoehto.

4.5 Testaus

Edellä esitettyä täsmäysalgoritmia (4.4.4 Uusi paranneltu täsmäysalgoritmi C++:lla sivulta 33 alkaen) aiotaan testata niin, että tehdään täsmäystestejä muistissa olevalle taulukolle, jonka koko on 32 megatavua, aakkoston koko 256 ja joka sisältää testiajoissa satunnaista dataa. Haettavien yhtäaikaisten satunnaisten hahmojen määrä tulee olemaan 1, 5, 10, 50, 100, 1000, 2000, 4000, 8000, 10000, 20000, 40000, 80000 sekä 100000 kappaletta. Testi on tarkoitus tehdä 20 kertaa kullekin hahmojen lukumäärälle niin, että hahmot arvotaan uudelleen aina vain joka 5. kerta, ja laskea keskiarvot. Jollain toisella algoritmilla myös varmistetaan, että myös kaikki haettavat hahmot todellakin löytyivät. Tällaisena yksinkertaisempana vertailukohta-algoritmina myös suorituskyvyn suhteen tulee olemaan täsmäysalgoritmi, joka on esitetty edellä kohdassa 4.3.4 Perustäsmäysalgoritmi toteutettuna C++:lla alkaen sivulta 25.

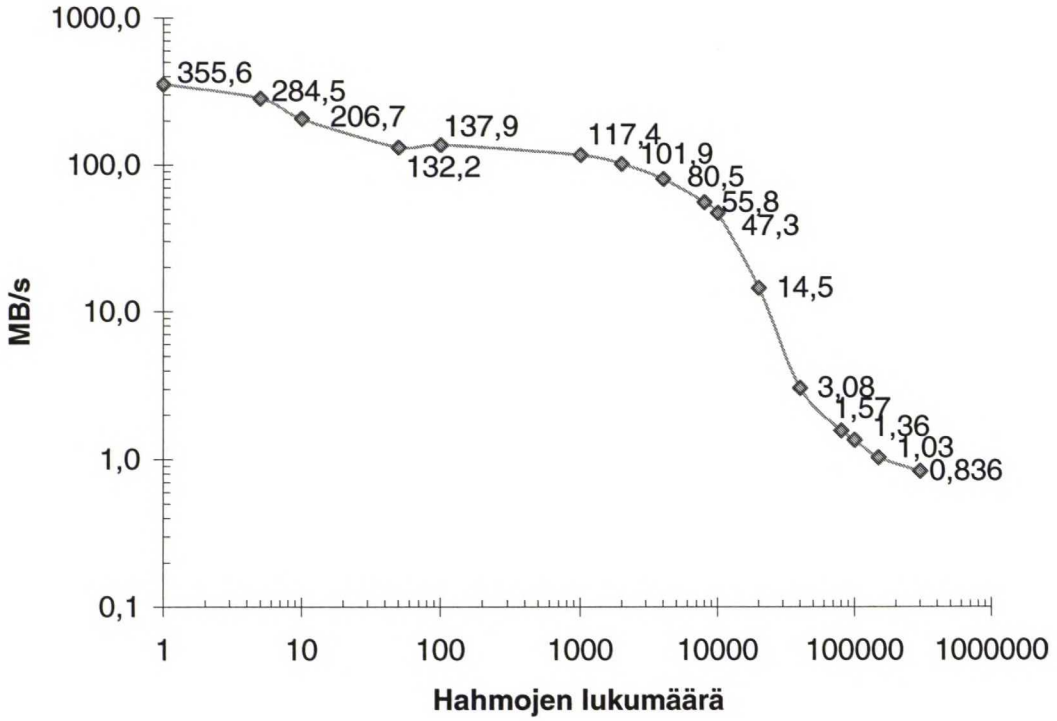
Testaamiseen käytetty kaikki koodi esitellään liitteessä A alkaen sivulta 54, mikäli joku on kiinnostunut toistamaan näitä testejä omassa laiteympäristössänsä ja vertailemaan tuloksia.

5. Tulosten tarkastelu

5.1 Uuden täsmäysalgoritmin nopeuden arviointia

Uuden täsmäysalgoritmin (4.4.4 Uusi paranneltu täsmäysalgoritmi C++:lla sivulta 33 alkaen) nopeuden arviointi suoritettiin lähinnä Intel-prosessoripohjaisella laitteistolla, jotta tulosten analysointi ja jatkokäsittely olisi helpompaa. Testikone oli HP Omnibook kannettava Celeron 650 MHz 128 MB muistilla.

Kyseinen testikone pystyi lukemaan ja kirjoittamaan testissä olleeseen 32 MB puskuriin noin 150 MB/s, mikä on hyvä vertailukohta arvioitaessa tämän uuden täsmäysalgoritmin tehokkuutta. Alla esitellään ehkäpä tärkein havainto täsmäysalgoritmin nopeudesta hahmojen lukumäärästä riippuen. Kaikki data on yksityiskohtaisemmin esiteltynä liitteessä B sivulta 58 alkaen.



Kuva 2. Täsmäysnopeuden riippuvuus samanaikaisten hahmojen lukumäärästä, kun hahmot ovat 8 tavun satunnaisista sormenjäljistä muodostettuja ja hakupuskurin koko on 32 MB ja sisältää satunnaista dataa.

On selvästi havaittavissa, että algoritmi toimii sangen tehokkaasti vielä niinkin suurella samanaikaisten hahmojen lukumäärällä kuin 10 000. Huomioimisen arvioista testi-

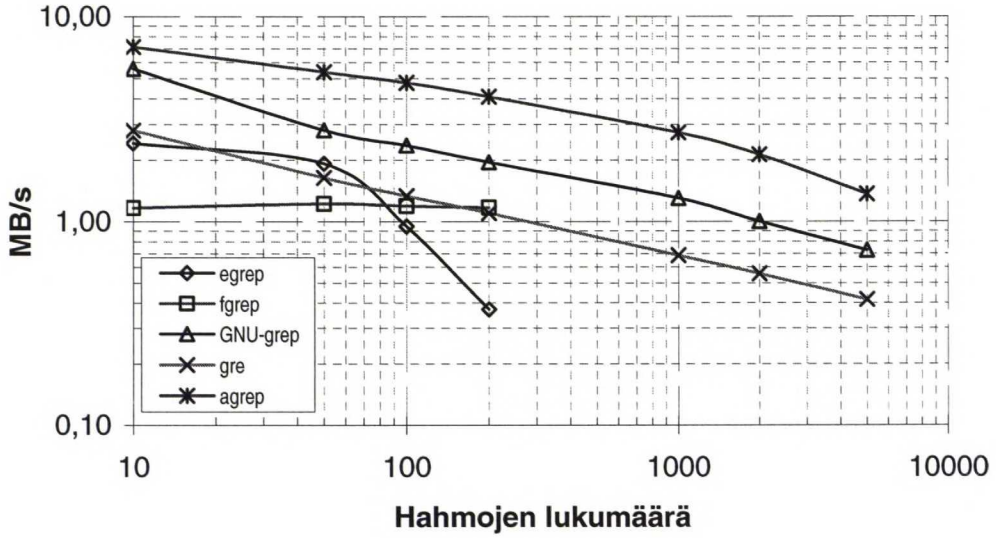
datasta on myös se seikka, että noin hahmojen lukumäärällä 180 kohdassa 4.4.2.1 *Eitäsmänneen viimeisen merkin sääntö* (sivulta 29 alkaen) esitetty algoritmin parannustapa menettää etunsa, jolloin se kannattaisi jättää pois koko lopullisesta algoritmista, mikäli tiedetään jatkuvasti käsiteltävän vähintään näin monia hahmoja. Samoin on havaittavissa, ettei sama parannustapa auta yhtään algoritmin loppupuolella suoritettaessa RK-mukaista uuden hajautusarvon laskentaa. Mielenkiintoista on myös havaita, että 100 hahmolla suorituskyky on jopa hieman parempi kuin 50 hahmolla!

Saadut tulokset ovat melko rohkaisevia jopa verrattaessa esim. tuloksiin, joita Josué Kuri ja Gonzalo Navarro ovat saaneet [20]. Heidän laitteistonsa tutkimuksissa oli ollut Sun Enterprise 450 serveri (4 x UltraSPARC-II 250 MHz) 512 MB muistilla ja saavutettu tulos 4 MB/s 100 samanaikaisella hahmolla, tosin hahmojen pituus on ollut tässä tapauksessa vain 4 merkkiä ja haku likimääräistä sallien jopa 4 merkin lisäämisen mihin väliin tahansa, mikä on huomioitava tuloksia verrattaessa. Vasta kun tämän diplomityön algoritmista on kehitetty vastaavanlaisen likimääräisyyden salliva versio, ei tuloksia pidä vertailla missään nimessä suoraan keskenänsä! Jatkossa heidän tulostansa voidaan käyttää hyvänä vertailukohtana.

Parempana vertailukohtana voitaisiin pitää tuloksia, joita SunWu ja Udi Manber ovat saaneet omissa tutkimuksissansa [33]. Taulukossa 1 (alla) esitellään heidän algoritmiansa käyttävän *agrep*-ohjelma verrattaessa neljään muuhun tunnettuun Unix-hakutyökaluun. Testilaitteistona oli Sun SparcStation 10 malli 510, jonka käyttöjärjestelmänä oli Solaris, ja kokeet ovat 10 toistokerran keskiarvoja. Testiaineistona oli 15,8 MB tekstitiedosto Wall Street Journalin artikkeleista, joista myös testihahmot poimittiin, jolloin myös kaikki hahmot esiintyivät haettavassa tekstissä.

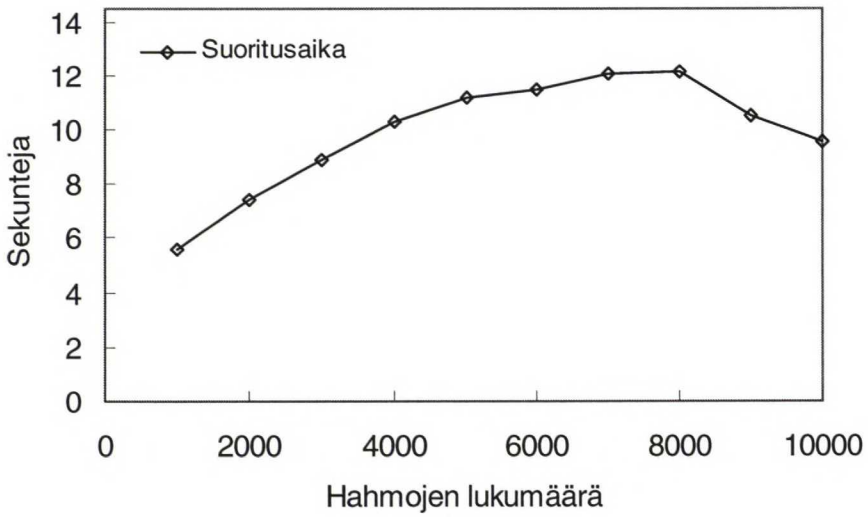
Hahmojen lukumäärä	egrep	fgrep	GNU-grep	gre	agrep
10	6,54	13,57	2,83	5,66	2,22
50	8,22	12,95	5,63	9,67	2,93
100	16,69	13,27	6,69	11,88	3,31
200	42,62	13,51	8,12	14,38	3,87
1000	-	-	12,18	23,14	5,79
2000	-	-	15,80	28,36	7,44
5000	-	-	21,82	38,09	11,61

Taulukko 1. Viiden eri hakualgoritmin vertailua 15,8 MB tekstitiedostolla, kun hahmojen koko on vaihteleva 5-15 merkkiä keskiarvokoon ollessa hiukan yli 6. Ajat ovat sekunteja. [33]



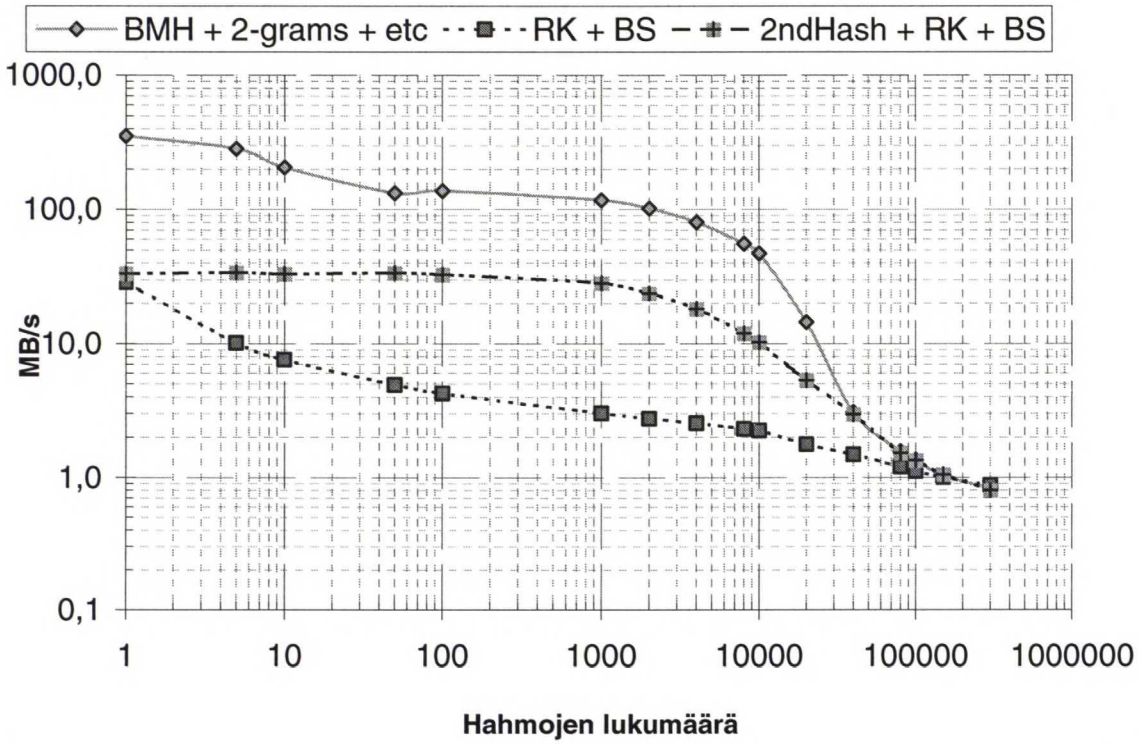
Kuva 3. Edellä esitetyn taulukon 1 informaatio esitettynä vastaavana täsmäysnopeutena.

Wu ja Manber vertailivat myös tarkemmin hahmojen lukumäärän ja suoritusajan suhdetta oman agrep-ohjelmansa osalta. Huomioitavaa alla esitettävässä kuvaajassa on se, että noin 8000 hahmon kohdalla suoritusajat putoavat, koska tämä on grep-työkalujen normaalin toimintatavan seurausta, sillä ne tulostavat vain kerran ne rivit, joilla hahmoja on esiintynyt. Kun yksi hahmo on löytynyt joltain riviltä, ei ole tarvetta tutkia riviltä muita hahmoja, vaan koko rivi tulostetaan ja siirrytään tutkimaan seuraavaa riviä.



Kuva 4. Suoritusajan riippuvuus samanaikaisten hahmojen lukumäärästä ***agrep****-hakutyökalulla. [33]*

Tässä diplomityössä esitetyn uuden täsmäysalgoritmin suorituskykyä (ylin kuvaaja alla olevassa kuvassa) vertailtiin hahmojen lukumäärän suhteen lopuksi myös kohdan 4.3.4 *Perustäsmäysalgoritmi toteutettuna C++:lla* (sivulta 25 alkaen) mukaiseen toteutukseen (alin kuvaaja) sekä tästä hieman paranneltuun versioon, jossa oli lisätty mukaan vain kohdan 4.4.1 *Kaksitasoinen hajautus* (sivulta 28 alkaen) toiminnallisuus (keskimmäinen kuvaaja).



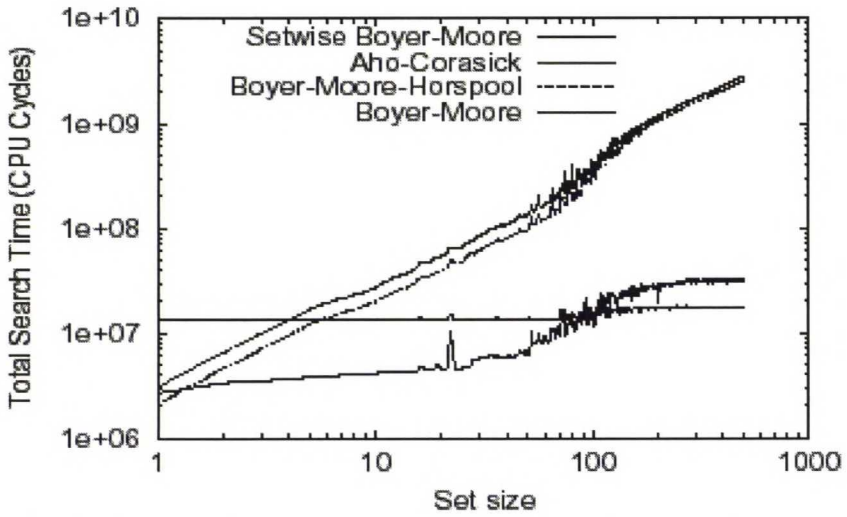
Kuva 5. Täsmäysnopeuden riippuvuus samanaikaisten hahmojen lukumäärästä, kun hahmot ovat 8 tavun satunnaisista sormenjäljistä muodostettuja ja hakupuskurin koko on edelleen 32 MB. Vertailussa kolme erilaista jo aikaisemmin esiteltyä algoritmia.

Havaintona voidaan sanoa suorituskykyeron olevan parhaan ja toiseksi parhaan algoritmin välillä 3 – 10–kertainen aina noin 20 000 hahmon lukumäärään saakka, jolloin 2-grammit eivät juurikaan enää pysty nopeuttamaan täsmäysalgoritmin toimintaa. Mentäessä yli 150 000 hahmon lukumäärän muuttuu perustoteutus, jossa on käytössä vain RK-algoritmi yhdistettynä binäärihakuun, nopeimmaksi algoritmiksi.

Aivan tämän diplomityön loppuvaiheessa löydettiin Mike Fiskin ja George Varghesen kaksi hyvää yhteistä artikkelia [10, 11], joiden kuvaamiin tuloksiin ei ehditty tehdä perusteellisia vertailuja. Tämä jää näin ollen ehdottomasti jatkokehitystyön kohteeksi. Pikaisesti verrattuna tämän diplomityön algoritmin pitäisi olla aivan kilpailukykyinen –

jollei jopa parempi – noissa artikkeleissa esitettyihin ratkaisuihin. Valitettavasti tarkempi analyysi jää nyt puuttumaan ajan takia.

Mainittakoon kuitenkin heidän tutkimustuloksistansa seuraavaa. Heidän tutkimustensa havainnoissa on ainakin eräs tärkeä yksityiskohta. Kun yhtäaikaisten hahmojen lukumäärä on alle 100, ovat BMH-tyyliset algoritmit yleensä AC-algoritmia nopeampia, kunhan vain itse toteutustapa on valittu huolella. He eivät kuitenkaan ole tutkineet, kuinka kirjainpareja ja 2-grammitekniikkaa voitaisiin hyödyntää merkkijono-täsmäyksessä, kuten juuri tämän diplomityön yhteydessä tehtiin. Tästä syystä tulosten vertaaminen nopeasti keskenään on hieman hankalahkoa.



Kuva 6. Fiskin ja Varghesen tutkimustuloksia nopeasti esiteltynä, mikä on heidän toteuttamansa C-kirjaston suorituskyky, prosessorin kellojaksojen lukumäärä riippuen yhtäaikaisten hahmojen lukumäärästä. Myös heidän algoritminsa (Setwise-Boyer-Moore-Horspool, SBMH) pohjautuu BMH-algoritmin pääideoihin, mutta se ei hyödynnä esim. 2-grammitekniikkaa. Huomaa, että kuvaajat ovat todellisuudessa päinvastaisessa järjestyksessä mm. 10 hahmon kohdalla kuin mitä on esitetty kuvan yläosan "merkkien selitys" –kohdassa (legend). [11]

5.2 Sormenjälkien valinnasta

Jotta tämän diplomityön täsmäysalgoritmi toimisi mahdollisimman tehokkaasti, kannattaa mahdollisia sormenjälkiä miettiessä kiinnittää huomiota seuraaviin asioihin. Etsittäessä sopivaa 8 tavun merkkijonoa kannattaisi suosia sellaisia, joissa esiintyy mahdollisimman paljon samanlaisia aakkoston merkkejä kuin muissakin hahmoissa tai jopa samoja merkkipareja. Myös etenkin löydettäessä sormenjälkiä, joiden alkuosat ovat

samanlaisia, muodostuu tilanne paremmaksi BMH-tyylisesti hyppyjen suhteen, sillä silloin hypyt olisivat useammin mahdollisimman pitkiä. Tässä hyvien sormenjälkien valintatekniikassa voisi myös olla yksi tärkeä jatkotutkimuksen alue.

5.3 Toiminnallisten vaatimusten tulokset

Hakumoottorin suunnittelun lähtökohtana ja suunnittelupäätösten perustana toimi ennen suunnittelua suoritettu laaja vaatimusten määrittely. Näiden pohjalta lähdettiin tutkimaan mahdollisia eri toteutustapoja ja –tekniikoita. Erityisesti kiinnitettiin huomiota valittavaan täsmäysalgoritmiin, jotta kaikki muut vaatimukset saataisiin myös toteutettua hyvin sen pohjalta. Tärkein työn tulos tässä vaiheessa oli kutsurajapinta hakumoottorin käyttöön.

5.4 Laitteistotekniset vaatimustulokset

Syntyneen hakumoottoriohjelmakoodin kannettavuudesta (*porting*) voidaan mainita lyhyesti sellaista, että sama lähdekoodi on saatu onnistuneesti kääntymään ja toimimaan jo useammassa käyttöjärjestelmäympäristössä. Näin olen yksi sama hakumoottori voisi olla näissä ympäristöissä Anti-Virus-ohjelmiston tiedostojen tutkimista suorittavana komponenttina. Vain tietokantojen sisällöt saattavat vaihdella sen mukaan, halutaanko kaikkia samoja viruksia tunnistaa molemmissa ympäristöissä vai ei.

Ohjelmabinäärin koko on lähes 50 kB, mutta samalla on huomioitava, että samaan sisältyy koodi, joka mahdollistaa pakattujen zip- ja jar-tiedostojen tutkimisen suoraan halutulle syvyystasolle saakka ilman, että tarvitsee luoda mitään väliaikaistiedostoja. Tämän ohjelmakoodin koko binäärissä on alle 8 kB ja yksi rekursiotaso pakatun tiedoston tasolla vaatii vain 40 kB lisämuistia jokaista tasoa kohti. On siis mahdollista tutkia tiedostoja vaikkapa kymmenen kertaa sisäkkäin pakatun zip-tiedoston kohdalla, jolloin muistia tarvitaan zipin läpikäymiseen 400 kB ja silti tutkiminen on melko nopeaa.

Ohjelmakoodi ei vaadi paljoakaan säikeen prosessoripinoa käyttöönsä. Voidaan olettaa, että useinkaan hakumoottorin toimesta pinoa ei ole varattuna 250 tavua enempää, vaan kaikki muistin varaamiset on pyritty tekemään dynaamisesta muistista. Samalla missään ei ole käytetty rekursiivista ohjelmointityyliä.

Muista aikaisemmin esitetyistä laitteistoteknisistä vaatimuksista voidaan sanoa, että hakumoottori toteuttaa ne aivan niin kuin pitääkin.

5.5 Suorituskyvyn ja käytettävyyden vaatimustulokset

Edellä esitetyt tulokset ja arviot uuden täsmäysalgoritmin nopeudesta osoittavat hakumoottorin täyttävän melko helposti sille osoitetut vaatimukset suorituskyvyn suhteen. Suurten tiedostojen tutkimisen hitauteen on varauduttu oletusarvoisesti 5 sekunnin aikarajoituksella. Jos tiedosto on niin suuri, ettei sitä jostain syystä ole saatu tutkittua kokonaan tuon 5 sekunnin aikana, keskeytyy tutkiminen siten, että tiedostosta tutkitaan lisäksi aivan viimeiset 8 kilotavua ja ilmoitetaan tulos tämän siihen saakka suoritetun tutkimisen pohjalta.

Suoritettaessa pikaisia käytettävyystestauksia eri kämmenmikrolaitteilla ei havaittu mitään suuria käytettävyysongelmia, jotka olisivat johtuneet juuri hakumoottorin toteutuksesta, kun Anti-Virus-ohjelmiston reaaliaikainen tutkiminen oli aktivoituna.

5.6 Muistinkulutus

Muistinkulutus on melko pientä täsmäysalgoritmin osalta. Hahmojen osalta vaaditaan, että jokaisesta 8-merkkisestä hahmosta pidetään muistissa puolet lyhyempi 4-merkkinen eli 32-bittinen hajautusarvo. Tämän lisäksi tarvitaan 256 tavua ei-täsmänneiden viimeisten merkkien taululle, 56 kB ei-täsmänneiden merkkiparien taululle ja 8 kB kaksitasoiseen hajautukseen ennen binäärihaun suorittamista. Työalueena tiedostoja tutkittaessa käytetään 4 kB kokoista työpuskuria. Näin ollen jopa 10 000 hahmon yhtäaikaiseen tutkimiseen riittää reilusti alle 128 kB vapaata muistia ja tarvittaessa jopa 100 000 hahmoa voidaan hakea vain noin 460 kB:lla. Tosin loput puoliskot hahmoista pitää olla tallennettuina jonnekin vaikkapa hitaampaan muistiin tai tiedostoon, jotta voidaan erottaa osuneet hajautusarvot varsinaisista oikeista täsmäyksistä ja lähelle osuneisiin "vääriin" arvoihin (*spurious hit*).

5.7 Jatkokehitys

Jatkokehityksestä tässä diplomityössä esitellylle hakumoottorille ei ole täyttä varmuutta, mutta seuraava suurempi askel voisi olla toteuttaa toimiva virusten puhdistus eli disinfektointi, jota ei tämän projektin puitteissa vielä otettu mukaan.

Täsmäysalgoritmitekniikan puolesta saattaa myös löytyä lisää tutkimusalueita. Olisi kiinnostavaa tietää, voidaanko täsmäystä parantaa tehokkaammaksi suurellakin hahmojen lukumäärällä, joka on enemmän kuin 10 000, siten, ettei kuitenkaan jouduta käyttämään kovin suuria muistimääriä. Tähän voisi löytyä ratkaisu melko helposti tutkimalla Wun ja Manberin tekniikkaa 3 merkin mittaisille alimerkkijonoille

hajautuksen kautta [33]. Toinen tutkimuskohde voisi olla, kuinka algoritmia pitäisi muokata, jotta se soveltuisi jopa likimääräiseen täsmäykseen, sallien tietyt määrät lisäyksiä, poistoja tai korvauksia hahmoihin. Lisäksi tutkimusaiheita voisivat olla, millainen vaikutus haun tehokkuuteen on valitulla hahmojen minimipituudella ja olisiko jopa järkevää pyrkiä käyttämään pidempiä sormenjälkiä, vaikkapa niinkin suuria kuin yli 30 tavua ja onko se edes käytännössä mahdollista ja järkevää?

Tutkimisen arvoista olisi myös, mitä yleensä tiedostoista kannattaa tutkia. Yleensä tietokonevirusinfektiot ovat olleet löydettävissä nopeasti joko tiedoston alussa tai lopussa, mutta välttämättä aina tilanne ei ole tämä sama.

On myös mahdollista löytää monia uusia sovelluskohteita tämän diplomityön algoritmille. Eräs mielenkiintoinen tutkimusalue Anti-Virus-ohjelmistojen ohella voisi olla erilaiset tunkeutumisen havaitsemisjärjestelmät (*Intrusion Detection Systems*), jotka ovat saaneet viime aikoina melko paljon huomiota maailmalla [10, 11, 24]. Lisäksi mm. tiedon louhinta (*Data Mining*) ja DNA-tutkimus (*DNA Searching*) voisivat myös hyötyä suurestikin tämän diplomityön tuloksista.

6. Yhteenveto

Tämän diplomityön tavoitteena oli suunnitella ja toteuttaa Anti-Virus-hakumoottori olennaiseksi ydinosaksi kämmenmikrolaitteiden Anti-Virus-ohjelmistolle. Työ on ollut osa suurempaa tuotekehityshanketta, jossa on tavoitteena parantaa Anti-Virus-ohjelmistojen saatavuutta etenkin langattomille laitteille. Samalla osoitettiin, että toimivan Anti-Virus-ohjelmiston toteuttaminen ole lainkaan mahdotonta nykyisille kämmenmikrolaitteille ainakaan hakumoottorin näkökulmasta.

Työn tuloksena syntyi varsin tehokas ratkaisu, joka tulee olemaan käyttökelpoinen ainakin niin kauan, ennen kuin tietokoneviruksia alkaa ilmaantua kämmenmikroihin niin paljon, että niiden puhdistaminen alkaa olla todellinen ongelma. Tosin silloinkin samaa hakumoottoria voidaan käyttää edelleen tunnistamiseen. Vasta sitten, kun tunnistettavien virusten määrä ja erityisesti virusperheiden määrä alkaa olla reilusti yli 10 000, pitää alkaa miettiä tehokkaampia uusia tunnistustekniikoita. Pienet virusvarianttien tunnistamiset voidaan toteuttaa muilla perinteisillä tarkistussummiin (*CRC*) perustuvilla tekniikoilla.

Seuraava iso ongelma-alue tulee ehkäpä olemaan polymorfiset virukset, joiden tehokkaaseen tunnistamiseen ei ole oikein muuta hyvää tunnettua tekniikkaa kuin koodin suorituksen emulointi. Polymorfisten virusten tunnistaminen tuleekin olemaan seuraava todella suuri haaste.

Hakumoottorin suunnittelutyö onnistui hyvin ja aivan aikataulun mukaisesti. Myös toteutusvaiheessa kaikki sujui aikataulussa ja usein jopa sitä hieman edellä. Suunniteltu hakumoottori täyttää hyvin myös asetetut vaatimukset ja erityisesti suorituskyky-vaatimukset ovat tulleet täytettyä erityisen hyvin, varsinkin huomioitaessa, kuinka pienellä muistin kulutuksella täsmäystekniikka on toteutettu, kuten viidennessä luvussa on testiajojen pohjalta todettu.

Tämän diplomityön aikana kehitetylle täsmäysalgoritmille on löydettävissä monia muitakin sovelluskohteita kuin vain Anti-Virus-ohjelmistot. Tällaisia sovelluskohteita voisivat olla mm. tunkeutumisen havaitsemisjärjestelmät, tiedon louhinta ja DNA-tutkimus. Työn tulokset ovat lähes suoraan sovitettavissa vanhempiinkin AV-ohjelmistoihin, jotka tällöin toimisivat entistä nopeammin.

7. Lähteet

[1] Aho, A. V. & Corasick, M. J., *Efficient string matching: an aid to bibliographic search.*, Communications of the ACM, Vol. 18(6), pp. 333-340, June 1975.

[2] Alaniemi T. & Oravainen S., *Symbian ja muistinhallinta*, Oulun Yliopisto — Tietojenkäsittelytieteiden laitos, 2001.
[Verkossa, viitattu 11.3.2002]
<URL:http://www.tol.oulu.fi/~antti/Ohjy/Raportit2001/Tuulikki%20Alaniemi%20ja%20Susanna%20Oravainen.pdf>

[3] Amoroso, E., *Fundamentals of Computer Security Technology*, Prentice-Hall, New Jersey, USA 1994. ISBN 0-13-108929-3

[4] Baeza-Yates, R. & Gonnet, G., *A New Approach to Text Searching.*, Communications of the ACM, Vol. 35(10), pp. 74-82, October 1992.

[5] Boyer, R. S., & Moore, J. S., *A Fast String Searching Algorithm*, Communications of the ACM Vol. 20(10), pp. 762-772, October 1977.

[6] Coit, C. J., Staniford, S. & McAlerney J., *Towards Faster String Matching for Intrusion Detection or Exceeding the Speed of Snort.*, in DARPA Information Survivability Conference and Exposition, Anaheim, CA, USA, June 12-14, 2001.

[7] Commentz-Walter, B., *A String Matching Algorithm Fast on the Average*, Proc. 6th International Colloquium on Automata, Languages and Programming, pp. 118-132, 1979.

[8] Computer Virus/alt FAQ, *[alt.comp.virus] FAQ.*
[Verkossa, viitattu 9.4.2002]
<URL:http://www.faqs.org/faqs/computer-virus/alt-faq/>

[9] Cormen, T. H., Leiserson, C. H. & Rivest R. L., *Introduction to Algorithms.*, MIT Press, Massachusetts, USA 1990. ISBN 0-262-03141-8

[10] Fisk, M. & Varghese, G., *Fast Content-Based Packet Handling for Intrusion Detection.*, UCSD Technical Report CS2001-0670, University of California, San Diego, USA, Department of Computer Science and Engineering, June 2001.

[11] Fisk, M. & Varghese, G., *An Analysis of Fast String Matching Applied to Content-Based Forwarding and Intrusion Detection.*, to appear in IEEE Infocom 2002 — The 21st Annual Joint Conference of the IEEE Computer and Communications Societies, New York, USA, June 23-27, 2002. (Successor to UCSD TR CS2001-0670).

[12] Frisk Software International, Home Page for F-Prot Antivirus. [Verkossa, viitattu 9.4.2002] <URL:http://www.f-prot.com/>

[13] Furber, S., *ARM System Architecture.*, Addison-Wesley Longman, Iso-Britannia 1996. ISBN 0-201-40352-8

[14] Furber, S., *ARM System-on-Chip Architecture.*, Addison-Wesley, Iso-Britannia 2000. ISBN 0-201-67519-6

[15] Gossin, S., Jones, N., McCurdy N. & Persaud, R., *Pattern Matching in Snort.*, University of California, San Diego, Department of Computer Science and Engineering, La Jolla, November 2001.

[16] Hepworth N., *Symbian OS — An Introduction*, Lecture material for HUT, Symbian Ltd., October 2001. [Verkossa, viitattu 11.3.2002] <URL:http://www.cs.hut.fi/~tarhio/pub/symbian.pdf>

[17] Horspool, N., *Practical Fast Searching in Strings.,* Software — Practice and Experience, Vol. 10(6), pp. 501-506, June 1980.

[18] Hume, A. & Sunday D., *Fast String Searching.*, Software — Practice and Experience, Vol. 21(11), pp. 1221-1248, November 1991.

[19] Knuth, D. E., Morris, J. H. & Pratt V. R., *Fast pattern matching in strings.*, SIAM Journal on Computing , Vol. 6(2), pp. 323-350, June 1977.

[20] Kuri, J., & Navarro, G., *Fast Multipattern Search Algorithms for Intrusion Detection*. In Proceedings of SPIRE'2000, pp. 169-180, 2000

[21] Lecroq, T. & Charras, C., *Exact String Matching Algorithms.*, Université de Rouen, Laboratoire d'Informatique de Rouen, Faculté des Sciences et des Techniques, France 2001. [Verkossa, viitattu 14.4.2001] <URL:http://www-igm.univ-mlv.fr/~lecroq/string/string.ps>

[22] Muth, R. & Manber, U., *Approximate Multiple String Search*, the 7th Annual Combinatorial Pattern Matching Symp., Laguna Beach, CA, USA, pp. 75-86. June 1996.

[23] Rabin, M. & Karp, R., *Efficient Randomized Pattern-Matching Algorithms.*, IBM Journal on Research Development Vol 31(2), pp.249-260, March 1987.

[24] Roesch, M., Snort — *Lightweight Intrusion Detection for Networks.*, Stanford Telecommunications, Inc., in Proceedings of the 13th Systems Administration Conference (LISA '99) pp. 229-238., USENIX. Seattle, Washington, USA, November 7–12, 1999.

[25] van Someren, A. & Atack, C., *The ARM RISC Chip — A Programmer's Guide.*, Addison-Wesley, Iso-Britannia 1993. ISBN 0-201-62410-9

[26] Sunday, D., *A Very Fast Substring Search Algorithm.*, Communications of the ACM, Vol. 33(8), pp. 132-142, August 1990.

[27] Sutinen, E., *Approximate Pattern Matching with the q-Gram Family.*, Ph.D. Thesis, Helsinki University Printing House, Helsinki 1998. ISBN 951-45-8250-0

[28] Tarhio, J. & Peltola, H., *String Matching in the DNA Alphabet.*, Software — Practice and Experience, Vol. 27(7), pp. 851-861, July 1997.

[29] Tarhio, J., *Henkilökohtaiset keskustelut ja tapaamiset.*, 2002.

[30] Tasker, M. et al., *Professional Symbian Programming — Mobile Solutions on the EPOC Platform.*, Wrox Press Ltd, Symbian Ltd, Iso-Britannia 2000. ISBN 1-861003-03-X

[31] Wright, C. A., Cumberland, L. & Feng, Y., *A Performance Comparison Between Five String Pattern Matching Algorithms.*, University of Southern Mississippi, USA, Dec 16, 1998. [Verkossa, viitattu 14.4.2001] <URL:http://ocean.st.usm.edu/~cawright/pattern_matching.html>

[32] Wu, S., *Approximate Pattern Matching and its Applications*, Ph.D. Thesis in Computer Science, University of Arizona, Tucson, USA, June 1992.

[33] Wu, S. & Manber, U., *A Fast Algorithm for Multi-Pattern Searching*, Technical Report TR-94-17, Department of Computer Science, University of Arizona, Tucson, USA, May 1994.

Liite A

Testaamisessa käytetty ohjelmakoodi, joka on toteutettu MS Windows-ympäristöön.

```
typedef unsigned char uint8;
typedef unsigned short int uint16;
typedef unsigned long uint32;
typedef unsigned int uint;
typedef signed char int8;
typedef signed short int int16;
typedef signed long int32;

#include <stdlib.h>
#include <memory.h>
#include <stdio.h>
#include <time.h>
#include <malloc.h>

//for little-endian processors, now using Intel
#define ENDIAN 0

class Scanner
{
public:
  Scanner() { };
  ~Scanner() { };

  void matchHashesTimed(int, const int);
  void identifyThis(uint32 const, const uint32*, const uint8*);
  void matchHashes(int, const int);
  void reset(void) { memset(analyseHits, 0, sizeof(analyseHits)); }

  uint8* signatureLo;
  uint32* hashTable32lo;
  uint32* hashTable32hi;
  uint8* badLastChar;
  uint8* badCharPair;
  uint8* hash2ndLevel;
  static const uint8 Scanner::andMask[8];
  unsigned int analyseHits[16];

  uint8 hashBfr[32*1024*1024+8];
};

const uint8 Scanner::andMask[8] = {
    0x01, 0x02, 0x04, 0x08, 0x10, 0x20, 0x40, 0x80 };

void Scanner::matchHashesTimed(int i, const int j)
{                              // start pos, end pos
  union {
    uint8  db[8];
    uint32 dw[2];
  } buffer;
  const uint8* pos = hashBfr - 1;
  const uint8* const limit = pos + j;
  pos += i;
  for(;;)
  {
    uint yy;
    do {        // Boyer-Moore-Horspool, Bad last char rule for any signature
      yy = *(pos += 8);
    } while(!badLastChar[yy]);
    uint8* tp = &buffer.db[7-ENDIAN*7];
    uint8* x2h = badCharPair + 7*0x2000;
    for(;;)
    {
      *tp = yy;
      uint xx = yy << 5;
      yy = *--pos;      // backing one step and make a comparison
      if(!((x2h-=0x2000)[(yy>>3)+xx]&andMask[yy&7]))    // 7th .. 1st 2-grams
        break;          // BMH, no signature can have this pair of chars
      if(limit < pos)   // Stop Condition Checking moved to here
        return;
      tp -= 1-2*ENDIAN;
      if(tp != &buffer.db[ENDIAN*7])
        continue;       // not yet at the start of potential match
      *tp = yy;         // now we have a very potential match
      uint32 hash = _lrotr(buffer.dw[1-ENDIAN], 1) ^ buffer.dw[0+ENDIAN];
      for(;;)
      {
        uint32 const b = hash ^ (hash >> 16);           // 2nd level hashing
        if(hash2ndLevel[((b>>3)&0x1FFF)]&andMask[b&7])  // a good candidate?
        {
          const uint32* top = hashTable32hi;
          const uint32* mid = hashTable32lo-1;
          while(top > mid)      // unless the hash was not in the database
```

```
          {                     // do the good old binary search
            uint32 const k = (top - ++mid) >> 1;
            uint32 const h = *(mid += k);
            if(h == hash)
            {
              //identifyThis(hash, mid, pos);     // scan for duplicates etc.
              break;
            }
            if(h > hash)
            {
              top = --mid;
              mid -= k;
            }
          }
        }
        // that hash wasn't in the database
        uint zz=*(pos += 8);
        if(!badLastChar[zz])     // Could a BMH step be possible?
          break;         // BMH, bad last char rule for any signature, pos+=8
        uint ww = (zz << 5) + 6*0x2000;
        zz = *--pos;
        if(!(badCharPair[(zz>>3)+ww]&andMask[zz&7])) // 7th 2-gram usable?
          break;         // BMH, no signature can have this substring, pos+=7
        if(limit < pos) // Remember to check the Stop Condition, too!
          return;
        pos -= 6;        // It could be much faster this way e.g. with Intel
        hash = _lrotr(*(uint32*)&pos[4], 1) ^ *(uint32*)&pos[0];
      }
      break;
    }
  }
}

void Scanner::identifyThis(uint32 const hash, const uint32* mid,
                           const uint8* pos)
{
  do {
    --mid;                      // scan for duplicates in the table
  } while(*mid == hash);        // and start with the first one!
  uint32 g = (++mid - hashTable32lo - 1) << 3;
  do {
    if(!memcmp(pos, signatureLo + (g += 8), 4))
      analyseHits[10]++;
    else
      analyseHits[11]++;
    uint32 hash2 = _lrotr(((uint32*)pos)[1], 1) ^ ((uint32*)pos)[0];
    if(hash2 != hash)
      analyseHits[9]++;
  } while(*++mid == hash);
}

void Scanner::matchHashes(int i, const int j)
{                       // start pos, end pos
  union {
    uint8  db[8];
    uint32 dw[2];
  } buffer;
  const uint8* pos = hashBfr - 1;
  const uint8* const limit = pos + j;
  pos += i;
  for(;;)
  {
    uint yy;
    do {        // Boyer-Moore-Horspool, Bad last char rule for any signature
      yy = *(pos += 8);
      if(!badLastChar[yy])
        analyseHits[0]++;
    } while(!badLastChar[yy]);
    uint8* tp = &buffer.db[7-ENDIAN*7];
    uint8* x2h = badCharPair + 7*0x2000;
    for(;;)
    {
      *tp = yy;
      uint xx = yy << 5;
      yy = *--pos;      // backing one step and make a comparison
      if(!((x2h-=0x2000)[(yy>>3)+xx]&andMask[yy&7]))    // 7th .. 1st 2-grams
      {
        analyseHits[7+((badCharPair-x2h)>>13)]++;
        break;          // BMH, no signature can have this pair of chars
      }
      if(limit < pos)   // Stop Condition Checking moved to here
        return;
      tp -= 1-2*ENDIAN;
      if(tp != &buffer.db[ENDIAN*7])
        continue;       // not yet at the start of potential match
      *tp = yy;         // now we have a very potential match
      uint32 hash = _lrotr(buffer.dw[1-ENDIAN], 1) ^ buffer.dw[0+ENDIAN];
      for(;;)
      {
        uint32 const b = hash ^ (hash >> 16);          // 2nd level hashing
```

```
        if(hash2ndLevel[((b>>3)&0x1FFF)]&andMask[b&7])  // a good candidate?
        {
          const uint32* top = hashTable32hi;
          const uint32* mid = hashTable32lo-1;
          while(top > mid)      // unless the hash was not in the database
          {                     // do the good old binary search
            uint32 const k = (top - ++mid) >> 1;
            uint32 const h = *(mid += k);
            if(h == hash)
            {
              identifyThis(hash, mid, pos);     // scan for duplicates etc.
              break;
            }
            if(h > hash)
            {
              top = --mid;
              mid -= k;
            }
          }
        }
        else
          analyseHits[8]++;
        // that hash wasn't in the database
        analyseHits[12]++;
        uint zz=*(pos += 8);
        if(!badLastChar[zz])     // Could a BMH step be possible?
        {
          analyseHits[13]++;
          break;        // BMH, bad last char rule for any signature, pos+=8
        }
        uint ww = (zz << 5) + 6*0x2000;
        zz = *--pos;
        if(!(badCharPair[(zz>>3)+ww]&andMask[zz&7])) // 7th 2-gram usable?
        {
          analyseHits[14]++;
          break;        // BMH, no signature can have this substring, pos+=7
        }
        analyseHits[15]++;
        if(limit < pos) // Remember to check the Stop Condition, too!
          return;
        pos -= 6;       // It could be much faster this way e.g. with Intel
        hash = _lrotr(*(uint32*)&pos[4], 1) ^ *(uint32*)&pos[0];
      }
      break;
    }
  }
}

int signCompare(const void *arg1, const void *arg2)
{
  uint32 hash1 = _lrotr(((uint32*)arg1)[1], 1) ^ ((uint32*)arg1)[0];
  uint32 hash2 = _lrotr(((uint32*)arg2)[1], 1) ^ ((uint32*)arg2)[0];
  if(hash1 < hash2)
    return -1;
  if(hash2 < hash1)
    return 1;
  return 0;
}

// The GetTickCount function retrieves
// the number of milliseconds that have elapsed since Windows was started.
extern "C" __declspec(dllimport) int __stdcall GetTickCount(void);

int main(int argc, char* argv[])
{
  if(argc < 3)
    return 1;
  Scanner* const engine = new Scanner;
  if(engine == NULL)
    return 2;
  else
  {
    unsigned int now = GetTickCount();
    for(int j = 0; j < 10; j++)
      memset(engine->hashBfr, 0, 32*1024*1024);
    double runtime = (double)(GetTickCount()-now)/10000;
    printf("Memory access time: %.3f s, %.3f MB/s\n",
      runtime, 33.554432f / runtime);
  }
  srand((unsigned)time(NULL));

  int patterns = strtoul(argv[1], NULL, 0);
  uint8* signatures = (uint8*) _alloca(8*patterns);
  int i = 8*patterns;
  while(--i >= 0)
    signatures[i] = (uint8) (rand() ^ time(NULL));
  qsort(signatures, patterns, 8, signCompare);

  engine->signatureLo = signatures;
  engine->hash2ndLevel = (uint8*) _alloca(0x2000+7*0x2000+0x100);
```

```
  memset(engine->hash2ndLevel, 0, 0x2000+7*0x2000+0x100);
  engine->badCharPair = engine->hash2ndLevel + 0x2000;
  engine->badLastChar = engine->hash2ndLevel + 0x2000 + 7*0x2000;
  engine->hashTable32lo = (uint32*) _alloca(sizeof(uint32)*(patterns+2));
  engine->hashTable32hi = engine->hashTable32lo++ + patterns;

  // the stop condition BMHHS
  memcpy(&engine->hashBfr[32*1024*1024], signatures, 8);
  // some sentinels just in case
  engine->hashTable32lo[-1] = 0xFFFFFFFFL;
  engine->hashTable32hi[1]  = 0x00000000L;
  for(i = 0; i < patterns; i++)
  {
    uint8* pos = signatures+i*8;
    uint32 hash = _lrotr(((uint32*)pos)[1], 1) ^ ((uint32*)pos)[0];
    engine->hashTable32lo[i] = hash;
    hash ^= (hash >> 16);
    engine->hash2ndLevel[((hash >> 3) & 0x1FFF)] |=
      Scanner::andMask[hash & 7];
    int k = 0;
    for(;;)
    {
      engine->badLastChar[pos[0]] = 1;
      if(k >= 7)
        break;
      uint xx = pos[0] + ((uint)pos[1] << 8);
      pos++;
      uint8* tables = &engine->badCharPair[((xx >> 3) & 0x1FFF)] +
        0x2000*k++ - 0x2000;
      uint8 mask = Scanner::andMask[xx & 7];
      do {
        *(tables += 0x2000) |= mask;
      } while(tables < &engine->badCharPair[7*0x2000-0x2000]);
    }
  }
  i = patterns - 1;
  while(--i >= 0)
    if(engine->hashTable32lo[i+1] < engine->hashTable32lo[i])
    {
      printf("Bad binary table!\n");
      delete engine;
      return 3;
    }

  int maxRounds = strtoul(argv[2], NULL, 0);
  printf("Results for %d patterns ", patterns);
  printf("using 32 MB random data buffer with %d rounds\n", maxRounds);
  struct
  {
    unsigned __int64 hits[16];
    double time;
  } timing;
  memset(&timing, 0, sizeof(timing));
  for(int round = 1; round <= maxRounds; round++)
  {
    srand((unsigned)time(NULL));
    i = 32*1024*1024;
    while(--i >= 0)
      engine->hashBfr[i] = (uint8) (rand() ^ round);
    engine->reset();
    engine->matchHashes(0, 32*1024*1024-8);
    printf("#%03d ", round);
    for(i = 0; i < 16; i++)
    {
      printf("%9d ", engine->analyseHits[i]);
      timing.hits[i] += engine->analyseHits[i];
    }
    unsigned int now = GetTickCount();
    engine->matchHashesTimed(0, 32*1024*1024-8);
    double runtime = (double)(GetTickCount()-now)/1000;
    printf("time: %.3f s\n", runtime);
    timing.time += runtime;
  }
  printf("Average:\n     ");
  for(i = 0; i < 16; i++)
    printf("%9d ", timing.hits[i] /= maxRounds);
  printf("time: %.3f s\n", timing.time /= maxRounds);
  printf("Throughput: %.3f MB/s\n", 33.554432f / timing.time);

  delete engine;
  return 0;
}
```

Liite B

Nopeustestin tulosdata pohjautuu liitteen A ohjelmakoodiin (sivulta 54 alkaen).

Taulukon sarakkeiden selitykset lyhyesti vasemmalta oikealle, 1. Pelkän muistilohkon, 32 MB käsittelyaika, kun kaikki tavut käydään lävitse, 2. vastaava nopeus MB/s, 3. – 10. BMH-tyylisten hyppyjen osumat käyttäen 2-grammeja apuna (kpl), 11. kaksitasoisen hajautusten nopeutusosumat välttäen turhaa binäärihakua (kpl), 12. bad = testi vain, ettei ole tyhmiä virheitä haussa (pitäisi olla aina 0), 13. löytyneet täydelliset sormenjälkitäsmäykset (kpl), 14. löytyneet samat hajautusarvot (kpl), mutta väärille eri sormenjäljille (*spurious hit*), 15. RK = Rabin-Karp-algoritmin hyödyntäminen yhdistettynä mahdolliseen binäärihakuun (kpl), 16. BMH-tyyliset hypyt (kpl) ulos 8 askeleella RK-algortimista, 17. samoin 7-askeleen hypyt (kpl), 18. Pelkän RK-hyödyntäminen (kpl) (16. + 17. + 18. = 15.), 19. täsmäysaika ko. hahmojen lukumäärälle 32 MB puskurissa, 20. täsmäysnopeus MB/s.

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.	17.	18.	19.	20.
memory access				BMH steps, using 2-grams														scanning speed	
										2nd level			same		BMH	step 7	pure		
s	MB/s	8	7	6	5	4	3	2	1	hashing	bad	match	hash	RK	RK	RK	RK	s	MB/s
1 (number of patterns)																			
0,179	178,771	4079026	131379	426	1	0	0	0	0	0	0	0	0	0	0	0	0	0,090	355,556
0,183	174,863	4078972	131433	432	2	0	0	0	0	0	0	0	0	0	0	0	0	0,090	355,556
0,181	176,796	4079205	131147	456	2	0	0	0	0	0	0	0	0	0	0	0	0	0,090	355,556
0,183	174,863	4079076	131302	448	2	0	0	0	0	0	0	0	0	0	0	0	0	0,090	355,556
0,182	176,323	4079070	131315	440,5	1,75	0	0	0	0	0	0	0	0	0	0	0	0	0,090	355,556
5																			
0,222	144,144	3654406	615089	2251	8	0	0	0	0	0	0	0	0	0	0	0	0	0,112	285,714
0,221	144,796	3653624	615966	2268	12	0	0	0	0	0	0	0	0	0	0	0	0	0,114	280,702
0,222	144,144	3654983	614446	2231	9	0	0	0	0	0	0	0	0	0	0	0	0	0,114	280,702
0,220	145,455	3698943	564170	2272	11	0	0	0	0	0	0	0	0	0	0	0	0	0,110	290,909
0,221	144,635	3665489	602418	2255,5	10	0	0	0	0	0	0	0	0	0	0	0	0	0,113	284,507
10																			
0,221	144,796	3108737	1236623	4678	19	0	0	0	0	0	0	0	0	0	0	0	0	0,162	197,531
0,219	146,119	3124768	1218356	4618	14	0	0	0	0	0	0	0	0	0	0	0	0	0,158	202,532
0,219	146,119	3247697	1077857	4627	17	0	0	0	0	0	0	0	0	0	0	0	0	0,150	213,333
0,221	144,796	3202353	1129755	4534	22	0	0	0	0	0	0	0	0	0	0	0	0	0,150	213,333
0,220	145,458	3170889	1165648	4614,25	18	0	0	0	0	0	0	0	0	0	0	0	0	0,155	206,682
50																			
0,219	146,119	1157670	3449323	24473	188	3	0	0	0	0	0	0	0	0	0	0	0	0,246	130,081
0,221	144,796	1227295	3369666	24567	198	0	0	0	0	0	0	0	0	0	0	0	0	0,242	132,231
0,221	144,796	1244248	3350451	24342	212	2	0	0	0	0	0	20	0	20	0	20	0	0,242	132,231
0,221	144,796	911373	3730436	24896	200	2	0	0	0	0	0	0	0	0	0	0	0	0,238	134,454
0,221	145,127	1135147	3474969	24569,5	199,5	1,75	0	0	0	0	0	5	0	5	0	5	0	0,242	132,249
100																			
0,220	145,455	260560	4452330	50072	633	8	0	0	0	0	0	0	0	0	0	0	0	0,232	137,931
0,221	144,796	149324	4579231	50334	636	6	0	0	0	0	0	0	0	0	0	0	0	0,230	139,130
0,221	144,796	149586	4579083	50155	640	5	0	0	0	0	0	0	0	0	0	0	0	0,232	137,931
0,221	144,796	204932	4516059	49875	653	8	0	0	0	0	0	0	0	0	0	0	0	0,234	136,752
0,221	144,961	191101	4531676	50109	640,5	6,75	0	0	0	0	0	0	0	0	0	0	0	0,232	137,936
1000																			
0,221	144,796	0	4377484	448570	41386	3257	209	12	0	0	0	0	0	0	0	0	0	0,272	117,647
0,222	144,144	0	4378313	448009	40931	3214	212	10	0	0	0	0	0	0	0	0	0	0,273	117,216
0,221	144,796	0	4376450	449431	41772	3297	204	9	0	0	0	0	0	0	0	0	0	0,274	116,788
0,220	145,455	0	4376009	449699	42078	3288	202	7	0	0	0	0	0	0	0	0	0	0,271	118,081
0,221	144,798	0	4377064	448927	41541,8	3264	206,75	9,5	0	0	0	0	0	0	0	0	0	0,273	117,433
2000																			
0,223	143,498	0	4000462	793295	139919	20908	2548	249	18	3	0	0	0	4	0	3	0	0,313	102,236
0,221	144,796	0	4001472	793096	138850	20823	2484	249	19	3	0	0	0	3	0	2	0	0,317	100,946
0,221	144,796	0	4002038	792594	138703	20777	2475	247	22	2	0	0	0	2	0	2	0	0,313	102,236
0,220	145,455	0	4002947	791993	138416	20449	2474	240	16	4	0	0	0	4	0	4	0	0,313	102,236
0,221	144,636	0	4001730	792745	138972	20739	2495,3	246,25	18,75	3	0	0	0	3,25	0	2,75	0	0,314	101,914
4000																			
0,222	144,144	0	3363422	1239315	405170	114017	26556	5066	913	327	0	0	0	349	0	226	122	0,398	80,402
0,220	145,455	0	3362695	1238226	406601	114974	26740	5042	922	349	0	0	0	368	0	242	126	0,394	81,218
0,220	145,455	0	3363647	1238597	405541	114184	26589	5069	896	337	0	0	0	360	0	242	118	0,404	79,208
0,222	144,144	0	3364904	1237713	405179	113823	26541	5033	978	325	0	0	0	349	0	229	120	0,395	81,013
0,221	144,799	0	3363667	1238463	405623	114250	26607	5052,5	927,25	334,5	0	0	0	356,5	0	234,75	121,5	0,398	80,460

1.	2.	3.	4.	5.	6.	7.	8.	9.	10.	11.	12.	13.	14.	15.	16.	17.	18.	19.	20.
memory access				BMH steps, using 2-grams						2nd level			same		BMH	step 7	pure	scanning speed	
s	MB/s	8	7	6	5	4	3	2	1	hashing	bad	match	hash	RK	RK	RK	RK	s	MB/s
8000																			
0,208	153,846		0 2416104	1554730	907261	464811	205057	80096	30402	24904	0	0	0	28138	0	11979	16159	0,575	55,652
0,204	156,863		0 2405068	1558563	912499	468326	207545	81199	31142	25826	0	24	0	29292	0	12327	16965	0,573	55,846
0,205	156,098		0 2411208	1559097	908309	465217	205578	80062	30695	24833	0	0	0	28116	0	11899	16216	0,577	55,459
0,207	154,589		0 2413614	1556734	908313	464520	205113	80202	30562	25031	0	0	0	28353	0	12057	16295	0,571	56,042
0,206	155,349		0 2411499	1557281	909096	465719	205823	80390	30700	25148,5	0	6	0	28474,75	0	12066	16408,75	0,574	55,750
10000																			
0,223	143,498		0 2051500	1551467	1069420	659901	359888	177135	84713	96268	0	0	11	112067	0	38242	73825	0,671	47,690
0,221	144,796		0 2050123	1548286	1070588	661137	362223	180595	85793	96944	0	0	0	113124	0	38510	74614	0,677	47,267
0,221	144,796		0 2050908	1553349	1069540	659177	359625	176859	84318	94778	0	0	0	110752	0	37984	72768	0,683	46,852
0,221	144,796		0 2064146	1542818	1069685	655370	356156	175508	83242	94132	0	0	0	109835	0	38002	71832	0,677	47,267
0,222	144,472		0 2054169	1548980	1069808	658896	359473	177524	84517	95530,5	0	0	2,75	111444,5	0	38185	73259,75	0,677	47,269
20000																			
0,221	144,796		0 806076	874825	943583	979955	974474	921778	775496	3423166	0	0	18	4646102	0	548088	4098014	1,991	16,072
0,222	144,144		0 800603	888895	954734	985974	980279	900698	759344	3390633	0	0	22	4610267	0	541058	4069209	2,127	15,045
0,221	144,796		0 797182	883346	941308	986564	979401	920320	765972	3422074	0	0	21	4664389	0	544579	4119809	2,500	12,800
0,221	144,796		0 798911	887053	939262	983147	975853	917987	772312	3425143	0	0	21	4662032	0	544705	4117326	2,267	14,116
0,221	144,633		0 800693	883530	944722	983910	977502	915196	768281	3415254	0	0	20,5	4645698	0	544608	4101090	2,221	14,508
40000																			
0,222	144,144		0 17818	27501	45794	80770	137388	228521	336357	15535167	0	0	334	28705310	0	400250	28305060	10,113	3,164
0,221	144,796		0 17692	27267	46477	80238	140879	227897	334952	15532047	0	0	256	28708650	0	398799	28309850	10,117	3,163
0,221	144,796		0 17097	26011	45480	79293	137322	223934	332191	15540052	0	0	251	28787398	0	393523	28393874	10,773	2,970
0,220	145,455		0 16723	25836	45107	77288	133714	220894	327991	15596010	0	0	262	28858488	0	388318	28470169	10,625	3,012
0,221	144,798		0 17332,5	26653,8	45714,5	79397	137326	225312	332873	15550819	0	0	275,75	28764962	0	395223	28369738	10,407	3,077
80000																			
0,185	172,973		0 3	6	18	84	249	853	2743	9723689	0	0	627	33498631	0	7156	33491474	19,875	1,610
0,191	167,539		0 0	1	2	3	14	51	152	9792532	0	0	652	33550510	0	510	33549999	20,474	1,563
0,185	172,973		0 2	3	21	77	235	863	2661	9738451	0	0	730	33498897	0	7139	33491758	20,413	1,568
0,188	170,213		0 1	7	20	66	244	772	2575	9707031	0	15452	587	33502595	0	6650	33495945	20,606	1,553
0,187	170,924		0 1,5	4,25	15,25	57,5	185,5	634,75	2032,8	9740426	0	3863	649	33512658	0	5363,8	33507294	20,342	1,573
100000																			
0,196	163,265		0 0	0	0	0	0	0	0	7237948	0	1	800	33554417	0	0	33554417	23,089	1,386
0,188	170,213		0 0	0	0	2	14	62	277	7217640	0	0	889	33546740	0	1030	33545710	23,455	1,364
0,189	169,312		0 0	0	0	3	15	61	249	7158787	0	0	800	33546821	0	1023	33545798	24,128	1,326
0,194	164,948		0 0	0	0	1	7	26	137	7143906	0	0	841	33550670	0	503	33550166	23,416	1,367
0,192	166,935		0 0	0	0	1,5	9	37,25	165,75	7189570	0	0,25	832,5	33549662	0	639	33549023	23,522	1,361
150000																			
0,187	171,123		0 0	0	0	0	0	0	0	3335798	0	0	1262	33554418	0	0	33554418	31,155	1,027
0,187	171,123		0 0	0	0	0	0	0	0	3335798	0	0	1174	33554418	0	0	33554418	31,155	1,027
0,187	171,123		0 0	0	0	0	0	0	0	3335798	0	0	1218	33554418	0	0	33554418	31,155	1,027
300000																			
0,175	182,857		0 0	0	0	0	0	0	0	336752	0	0	2640	33554418	0	0	33554418	38,295	0,836
0,175	182,857		0 0	0	0	0	0	0	0	336752	0	0	2666	33554418	0	0	33554418	38,285	0,836
0,175	182,857		0 0	0	0	0	0	0	0	336752	0	0	2653	33554418	0	0	33554418	38,290	0,836

Liite C

Vaihtoehtoinen tapa toteuttaa tässä diplomityössä esitetty täsmäysalgoritmi. Toisilla prosessoriarkkitehtuureilla tämäntapainen lähestymistapa saattaa olla nopeampi, koska käytännössä alla oleva koodi tuottaa vähemmän konekielikäskyjä. Nopeuserot ovat mahdollisia riippuen prosessorin cache-arkkitehtuurista.

```
void Scanner::matchHashes(int i, const int j)
{                               // start pos, end pos
  const uint8* pos = hashBfr - 1;
  const uint8* const limit = pos + j;
  const uint8* const t = badCharPair;
  pos += i;
  for(;;)
  {
    uint g,h;
    do {        // Boyer-Moore-Horspool, Bad last char rule for any signature
      h = *(pos += 8);
    } while(!t[h]);
    // 7th .. 1st 2-grams
    if(!(t[(h = (h << 8) + *--pos) + 256] & 0x01))
      continue;
    if(limit < pos)   // Stop Condition Checking moved to here
      return;
    if(!(t[(uint16) (h = (h << 8) + *--pos) + 256] & 0x02) ||
       !(t[(uint16) (h = (h << 8) + *--pos) + 256] & 0x04) ||
       !(t[(uint16) (g = (h << 8) + *--pos) + 256] & 0x08) ||
       !(t[(uint16) (g = (g << 8) + *--pos) + 256] & 0x10) ||
       !(t[(uint16) (g = (g << 8) + *--pos) + 256] & 0x20) ||
       !(t[(uint16) (g = (g << 8) + *--pos) + 256] & 0x40))
      continue;
    for(;;)
    {
      h = _lrotr(h, 1) ^ g;
      if(t[(uint16) (h ^ (h >> 16)) + 256] & 0x80)      // 2nd level hashing
      {                                                 // a good candidate?
        const uint32* top = hashTable32hi;
        const uint32* mid = hashTable32lo-1;
        while(top > mid)        // unless the hash was not in the database
        {                       // do the good old binary search
          uint32 const k = (top - ++mid) >> 1;
          uint32 const j = *(mid += k);
          if(j == h)
          {
            //identifyThis(hash, mid, pos);       // scan for duplicates etc.
            break;
          }
          if(j > h)
          {
            top = --mid;
            mid -= k;
          }
        }
      }
      // that hash wasn't in the database
      h = *(uint32*)&pos[5];
      pos += 7;
      if(!(t[(h >> 16) + 256] & 0x01))  // 7th 2-gram usable?
        break;          // BMH, no signature can have this substring, pos+=7
      if(limit < pos)   // Remember to check the Stop Condition, too!
        return;
      pos -= 6;
      g = *(uint32*)pos;
    }
  }
}
...
engine->signatureLo = signatures;
engine->badCharPair = (uint8*) _alloca(0x2000+7*0x2000+0x100);
memset(engine->badCharPair, 0, 0x2000+7*0x2000+0x100);
engine->hashTable32lo = (uint32*) _alloca(sizeof(uint32)*(patterns+2));
engine->hashTable32hi = engine->hashTable32lo++ + patterns;
...
for(i = 0; i < patterns; i++)
{
  uint8* pos = signatures+i*8;
  uint32 hash = _lrotr(((uint32*)pos)[1], 1) ^ ((uint32*)pos)[0];
  engine->hashTable32lo[i] = hash;
  hash ^= (hash >> 16);
  engine->badCharPair[256+(uint16) hash] |= 0x80;
  int k = 0;
  for(;;)
  {
    uint xx = pos[0];
    engine->badCharPair[xx] = 1;
    if(k >= 7)
      break;
    engine->badCharPair[256+xx+((uint)*++pos<<8)] |= (uint8) (0xFF >> ++k);
  }
}
...
```